Fachhochschule Dortmund

University of Applied Sciences and Arts

(Master Embedded Systems for Mechatronics)

ROS 2 QoS Settings Evaluation in Lossy Wireless Networks for UAVs

Research’s Thesis

Author: LE, Quang Hien

Matriculation Number: 7206974

Supervisor: Prof. Dr. Christof Röhrig

Co-Supervisor: Merlin Stampa

Date: 07/2021

# Abstract

Since Robot Operating System 1 (ROS 1) has been initially created in 2007 by Willow Garage, it has become one of the most common open-source robotics communities. Along with many advantages, it has also some disadvantages like not providing real-time control and robot security, ROS 2 has been created to leverage what is great about ROS 1 and providing what is not. The development of Unmanned Aerial Vehicle (UAV) is complicated and always requires real-time operation and non-ideal network handling when needed. A team of UAVs working together and communicating with the Ground Control Station (GCS) via a wireless network. In the field, wireless networks are usually lossy. While ROS 1 uses Transmission Control Protocol (TCP) as the basic transport, which is not suitable for lossy networks, ROS 2 has been developed to reply on Data Distribution Service (DDS) with User Datagram Protocol (UDP) as its transport, that can control the reliability of a node and act appropriately. The Quality of Service (QoS) allows software engineers to control this reliability and also other aspects. Here the QoS will be evaluated in a lossy wireless network. An experiment is set up using PX4-Fast RTPS(DDS) bridge which adds a Real Time Publish Subscribe (RTPS) interface to the open-source autopilot system for autonomous aircraft (PX4 Autopilot or PX4) enabling the communication between PX4 and ROS2. The experiment sends setpoints to the PX4 and also receives images and sensor data from PX4. The message ages, message periods as well as message loss will be observed and assessed. The result shows that with properly adjusting QoS settings, the setpoints messages to the PX4 which is the most important are all transmitted without a single message loss with a desirable delay, other messages transmission have also better performance.

Table of Contents

[Abstract ii](#_Toc77627725)

[List of Abbreviations vi](#_Toc77627726)

[List of Figures vii](#_Toc77627727)

[List of Tables x](#_Toc77627728)

[1. Introduction 1](#_Toc77627729)

[1.1. Motivation 1](#_Toc77627730)

[1.2. Problem statement 2](#_Toc77627731)

[1.3. Structure 3](#_Toc77627732)

[2. Background 3](#_Toc77627733)

[2.1. ROS 2 basics 3](#_Toc77627734)

[2.1.1. Nodes 3](#_Toc77627735)

[2.1.2. Topics 4](#_Toc77627736)

[2.1.3. Parameters 5](#_Toc77627737)

[2.1.4. Launch files 6](#_Toc77627738)

[2.1.5. Packages 7](#_Toc77627739)

[2.1.6. Workspace 8](#_Toc77627740)

[2.2. RTPS/DDS Interface: PX4-Fast RTPS(DDS) Bridge 8](#_Toc77627741)

[2.2.1. Architectural overview 9](#_Toc77627742)

[2.2.2. Code generation 10](#_Toc77627743)

[2.2.3. uORB messages support 10](#_Toc77627744)

[2.2.4. Client (PX4/PX4-Autopilot) 11](#_Toc77627745)

[2.2.5. Agent in an Offboard Fast DDS interface (ROS-independent) 12](#_Toc77627746)

[2.3. Quality of Service Settings 13](#_Toc77627747)

[2.3.1. QoS policies 13](#_Toc77627748)

[2.3.2. QoS profiles 14](#_Toc77627749)

[2.3.3. QoS compatibilities 15](#_Toc77627750)

[2.3.4. QoS events 15](#_Toc77627751)

[2.3. Evaluation tools 16](#_Toc77627752)

[2.3.1. Linux network traffic control utility 16](#_Toc77627753)

[2.3.2. ROS 2 Topic Statistics 16](#_Toc77627754)

[2.4. Related works 17](#_Toc77627755)

[3. Experimental Design and Setup 18](#_Toc77627756)

[3.1. Architecture of the experiments 18](#_Toc77627757)

[3.1.1. Unmanned aerial vehicles in fire fighting 18](#_Toc77627758)

[3.1.2. Architecture 19](#_Toc77627759)

[3.1.3. Package list in the workspace 20](#_Toc77627760)

[3.1.4. System setup 24](#_Toc77627761)

[3.2. Experiment with the default policy of QoS settings 24](#_Toc77627762)

[3.2.1. Introduction 24](#_Toc77627763)

[3.2.2. Implementation 25](#_Toc77627764)

[3.3. Experiment with the best\_effort policy of QoS settings 29](#_Toc77627765)

[3.3.1. Introduction 29](#_Toc77627766)

[3.3.2. Implementation 29](#_Toc77627767)

[4. Results Discussion 32](#_Toc77627768)

[4.1. Experiment with the default policy of QoS settings 32](#_Toc77627769)

[4.2. Experiment with the best-effort policy of QoS settings 43](#_Toc77627770)

[4.3. Comparison 53](#_Toc77627771)

[5. Conclusion 60](#_Toc77627772)

[5.1. Summary 60](#_Toc77627773)

[5.2. Future work 60](#_Toc77627774)

[5.2.1. Tuning of Additional QoS 60](#_Toc77627775)

[5.2.2. Use Case for a team of UAVs 60](#_Toc77627776)

[5.2.3. Evaluation QoS policies in real UAVs 61](#_Toc77627777)

[Bibliography 62](#_Toc77627778)

# List of Abbreviations

|  |  |
| --- | --- |
| ROS | Robot Operating System |
| RTPS | Real Time Publish Subscribe |
| DDS | Data Distribution Service |
| QoS | Quality of Service |
| TCP | Transmission Control Protocol |
| UAV | Unmanned Aerial Vehicles |
| UDP | User Datagram Protocol |
| GCS | Ground Control Station |
| uORB | Ubiquitous Object Request Broker |
| UART | Universal asynchronous receiver-transmitter |
| CDR | Common Data Representation |

# List of Figures

[Figure 1: Nodes - Topic and Service [4] 4](#_Toc77627645)

[Figure 2: Topic - Multiple Publisher and Multiple Subscriber [5] 5](#_Toc77627646)

[Figure 3: ROS 2 specific arguments in command line [7] 5](#_Toc77627647)

[Figure 4: ROS 2 configures argument using command line [8] 5](#_Toc77627648)

[Figure 5: Launch file example [10] 6](#_Toc77627649)

[Figure 6: How to run launch file [10] 7](#_Toc77627650)

[Figure 7: ROS 2 CMake packages with the minimum required contents [12] 7](#_Toc77627651)

[Figure 8: ROS 2 Python packages with the minimum required contents [12] 7](#_Toc77627652)

[Figure 9: Other directories in ROS 2 packages [11] 8](#_Toc77627653)

[Figure 10: A typical workspace in ROS 2 8](#_Toc77627654)

[Figure 11: microRTPS Bridge Architecture 9](#_Toc77627655)

[Figure 12: uorb\_rtps\_message\_ids.yaml file [13] 11](#_Toc77627656)

[Figure 13: Build PX4 SITL RTPS [13] 11](#_Toc77627657)

[Figure 14: Launching Client options [13] 12](#_Toc77627658)

[Figure 15: Starting the Client using UDP [13] 12](#_Toc77627659)

[Figure 16: The Agent command options [13] 12](#_Toc77627660)

[Figure 17: Starting the Agent using UDP [13] 12](#_Toc77627661)

[Figure 18: Installing iproute for tc 16](#_Toc77627662)

[Figure 19: Simulating 10% packet loss 16](#_Toc77627663)

[Figure 20: The architecture of the experiments 19](#_Toc77627664)

[Figure 21: List of packages in the workspace 20](#_Toc77627665)

[Figure 22: bridge\_msgs package structure 20](#_Toc77627666)

[Figure 23: image\_tools package structure 21](#_Toc77627667)

[Figure 24: launch\_bringup package structure 21](#_Toc77627668)

[Figure 25: operation\_control package structure 22](#_Toc77627669)

[Figure 26: px4\_msgs package structure 22](#_Toc77627670)

[Figure 27: px4\_ros\_com package structure 23](#_Toc77627671)

[Figure 28: System specification 24](#_Toc77627672)

[Figure 29: Advertiser is publishing Setpoints 25](#_Toc77627673)

[Figure 30: Subscribing Setpoint topic 26](#_Toc77627674)

[Figure 31: Publishing sensor data of the drone 27](#_Toc77627675)

[Figure 32: Subscribing Sensor data 28](#_Toc77627676)

[Figure 33: Image publisher [19] 28](#_Toc77627677)

[Figure 34: Image subscriber [19] 28](#_Toc77627678)

[Figure 35:QoS policies definition [19] 29](#_Toc77627679)

[Figure 36: GCS best\_effort launch 30](#_Toc77627680)

[Figure 37: Parsing parameters [19] 31](#_Toc77627681)

[Figure 38: QoS Initialization [19] 31](#_Toc77627682)

[Figure 39: Topic Statistics enabling 32](#_Toc77627683)

[Figure 40: Reliable output terminal of topic /statistics\_setpoint in 1% packet loss 33](#_Toc77627684)

[Figure 41: Statistics data\_type look-up table [23] 34](#_Toc77627685)

[Figure 42: Setpoint message age average in reliable policy from 0 to 9% packet loss 35](#_Toc77627686)

[Figure 43: Setpoint message period average in reliable policy from 0 to 9% packet loss 35](#_Toc77627687)

[Figure 44: Setpoint message age average in reliable policy from 10 to 50% packet loss 36](#_Toc77627688)

[Figure 45: Setpoint message period average in reliable policy from 10 to 50% packet loss 36](#_Toc77627689)

[Figure 46: Reliable output terminal of topic /statistics\_image in 1% packet loss 37](#_Toc77627690)

[Figure 47: Image message age average in reliable policy from 0 to 2% packet loss 39](#_Toc77627691)

[Figure 48: Image message period average in reliable policy from 0 to 2% packet loss 39](#_Toc77627692)

[Figure 49: Reliable output terminal of topic /statistics\_sensor in 1% packet loss 40](#_Toc77627693)

[Figure 50: Sensor message age average in reliable policy from 0 to 9% packet loss 41](#_Toc77627694)

[Figure 51: Sensor message period average in reliable policy from 0 to 9% packet loss 41](#_Toc77627695)

[Figure 52: Sensor message age average in reliable policy from 10 to 40% packet loss 42](#_Toc77627696)

[Figure 53: Sensor message period average in reliable policy from 0 to 40% packet loss 42](#_Toc77627697)

[Figure 54: Best\_effort output terminal of topic /statistics\_setpoint in 1% packet loss 43](#_Toc77627698)

[Figure 55: Best\_effort output terminal of offboard\_control node in 1% packet loss 44](#_Toc77627699)

[Figure 56: Setpoint message age average in best\_effort policy from 0 to 9% packet loss 45](#_Toc77627700)

[Figure 57: Setpoint message period average in best\_effort policy from 0 to 9% packet loss 45](#_Toc77627701)

[Figure 58: Setpoint message age average in best\_effort policy from 10 to 90% packet loss 46](#_Toc77627702)

[Figure 59: Setpoint message period average in best\_effort policy from 10 to 90% packet loss 47](#_Toc77627703)

[Figure 60: Best\_effort output terminal of topic /statistics\_image in 1% packet loss 48](#_Toc77627704)

[Figure 61: Image message age average in best\_effort policy from 0 to 2% packet loss 49](#_Toc77627705)

[Figure 62: Image message age average in best\_effort policy from 0 to 2% packet loss 49](#_Toc77627706)

[Figure 63: Best\_effort output terminal of topic /statistics\_sensor in 1% packet loss 50](#_Toc77627707)

[Figure 64: Sensor message age average in best\_effort policy from 0 to 9% packet loss 51](#_Toc77627708)

[Figure 65: Sensor message period average in best\_effort policy from 0 to 9% packet loss 51](#_Toc77627709)

[Figure 66: Sensor message age average in best\_effort policy from 10 to 90% packet loss 52](#_Toc77627710)

[Figure 67: Sensor message period average in best\_effort policy from 10 to 90% packet loss 52](#_Toc77627711)

[Figure 68: Setpoint message age average from 0 to 9% packet loss 54](#_Toc77627712)

[Figure 69: Setpoint message period average from 0 to 9% packet loss 54](#_Toc77627713)

[Figure 70: Setpoint message age average from 10 to 90% packet loss 55](#_Toc77627714)

[Figure 71: Setpoint message period average from 10 to 90% packet loss 55](#_Toc77627715)

[Figure 72: Image message age average from 0 to 2% packet loss 56](#_Toc77627716)

[Figure 73: Image message period average from 0 to 2% packet loss 56](#_Toc77627717)

[Figure 74: Sensor message age average from 0 to 9% packet loss 58](#_Toc77627718)

[Figure 75: Sensor message period average from 0 to 9% packet loss 58](#_Toc77627719)

[Figure 76: Sensor message age average from 10 to 90% packet loss 59](#_Toc77627720)

[Figure 77: Sensor message period average from 10 to 90% packet loss 59](#_Toc77627721)

# List of Tables

[Table 1: QoS policy setting options 14](#_Toc77627722)

[Table 2: QoS compatibilities of reliability and durability 15](#_Toc77627723)

[Table 3: QoS events 16](#_Toc77627724)

# 1. Introduction

## 1.1. Motivation

To create a development environment for the PR2 robot, Willow Garage has started developing ROS. It is aimed to provide the software tools that users can take to undergo research and production development of PR2. However, there is not only PR2 in the world, so the team of ROS had decided to support other robots as well and then made a great effort to create software interfaces that allow software as much as possible to be used elsewhere.

ROS has been successfully fulfilled requirements to provide the software tools for PR2 in the use case of a single robot, with no real-time requirements, excellent network connectivity, and applications in mostly academia. Nonetheless, when adapting to a variety of robots like wheeled robots, legged humanoids, industrial arms, outdoor ground vehicles, aerial vehicles, ROS has shown some disadvantages with other new use cases.

Furthermore, a huge number of robots used ROS in the industry alongside the academic research which is the initial focus. ROS-based products are coming to market, including manufacturing robots, agricultural robots, commercial cleaning robots, and others [1]. ROS has also been used by government agencies.

With the new use cases of the broader ROS community including teams of multiple robots, small embedded platforms, real-time systems, non-ideal networks, production environments, prescribed patterns for building and structuring systems, there should be some changes to adapt with these use cases.

At the core of ROS is an anonymous publish-subscribe middleware system that is built almost entirely from scratch [1]. Since 2007, there has been the development, adoption of several new technologies that are relevant to ROS such as Zeroconf, Protocol Buffers, ZeroMQ (and the other MQs), Redis, WebSockets, and DDS. Now, ROS should be rebuilt like a middleware system using existing open-source libraries so that it can be less code, taking a lot of features from open source libraries.

It has been several years, API has been changed a lot. Although the current APIs are still stable, there are not the best for ROS. So the new ROS, aka ROS 2, will have new APIs which are sophisticated of the community.

From the first alpha release of ROS 2 was in 2015 to the latest version of ROS 2 Galactic Geochelone, ROS 2 has gained popularity and ROS developers have started to discover ROS 2. As a result, ROS 2 evaluation should be carried out to obtain a full understanding of all the great features of ROS 2 that ROS 1 does not have. The development of UAV carries many struggles and difficulties. A team of UAVs is also required to do specific tasks, for example, mapping and navigation. This requires communication between them and GCS as well. Therefore, reliable communication should be needed to fulfill tasks in wireless networks which are usually non-ideal and lossy. ROS 2 offers QoS features, that give control over the level of reliability of a node, which helps to maintain better communication in UAVs.

## 1.2. Problem statement

Over the past 5 years, along with the arrival of ROS 1, the development of UAV has been grown significantly. At the moment, ROS 1 cannot meet the new use cases of UAVs, for example, a group of UAVs working together, real-time deadlines, and non-ideal networks condition. One of the ongoing projects that use ROS 1, which has UAV swarms for fire and rescue missions. During the missions, there is the case that one of the UAVs will flight behind the wall of the rescue building. As a result, the communication between this UAV and GCS will not stable and lossy. Therefore, the UAV cannot be received or lately received the commands from GCS. Consequently, the missions will be failed. That is because ROS 1 uses TCP as the main transport, which is unable to handle lossy wireless networks. The success of the rescue mission is vital. At the building on fire, there are many people needed to save, if the mission has failed, it will be horrible consequences. Due to the lack of important features, ROS 2 has arrived with the promise to cover the non-ideal network's case. This research aims to investigate the QoS feature of ROS 2 in the context of UAV swarms for fire and rescue missions during non-ideal networks. It will identify a set of policies in QoS profiles and conduct experiments to measure the effectiveness of the QoS policies.

## 1.3. Structure

Chapter 2, Background, describes features ROS 2, PX4-Fast RTPS(DDS) bridge, QoS policies, evaluation tools, and related work. Chapter 3, Experiment Implementation, explains how ROS 2 had been evaluated. Chapter 4, Results Discussion, reviews the outcome of the experiments. Chapter 5, Conclusion, gives the summary of the results, the answers to the problem statement, and the outlook of possible continuing research.

# 2. Background

ROS is an open-source robotics middleware including tools, libraries, and conventions that simplify the work of creating robust and complex robots in different robotic platforms. Although ROS is not an operating system but a collection of software frameworks for robot software development, it provides services designed for a heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management [2]. ROS is not a real-time Operating System, but it is possible to integrate ROS with real-time code [3].

Code reuse in robotics research and development is the essential goal of ROS. ROS is a distributed framework of processes (aka Nodes) that enables executables to be individually designed and loosely coupled at runtime [3]. These nodes can be organized together into Packages, which can be quickly shared and distributed.

With more than 5 years old, ROS 1 had succeeded their role in the research and development of robots. However, now sophisticated robots require more than ROS 1 can offer. ROS 1 cannot support the new cases, for example, a group of robots, real-time control, non-ideal networks. Therefore, ROS 2 has come to inherit what is good in ROS 1 and provide what is missed in ROS 1. This chapter will introduce the basis of ROS 2, RTPS/DDS interface, QoS policies, evaluation tools as well as related works.

## 2.1. ROS 2 basics

### 2.1.1. Nodes

A node is a process that performs computation [4]. Nodes are combined into a graph and communicate with one another using topics, services, actions, or parameters. Figure 1 below shows that Nodes are publishing and subscribing messages through topics as well as sending a request and receiving a response through service. The messages can include data, commands, or other information necessary for the system. These nodes serve as the executable code. A robot control system will usually consist of a number of nodes. For example, one node analyzes sensor data, one node controls images, one node controls wheel motors, and so on. The nodes can be entirely on one computer, or nodes can be distributed between computers or between computers and robots. Nodes usage offers several benefits to the system such as fault tolerance and code complexity. Crashes will be isolated to individual nodes. Implementation details are separated and hidden as the nodes only disclose a minimal API to the rest of the graph.

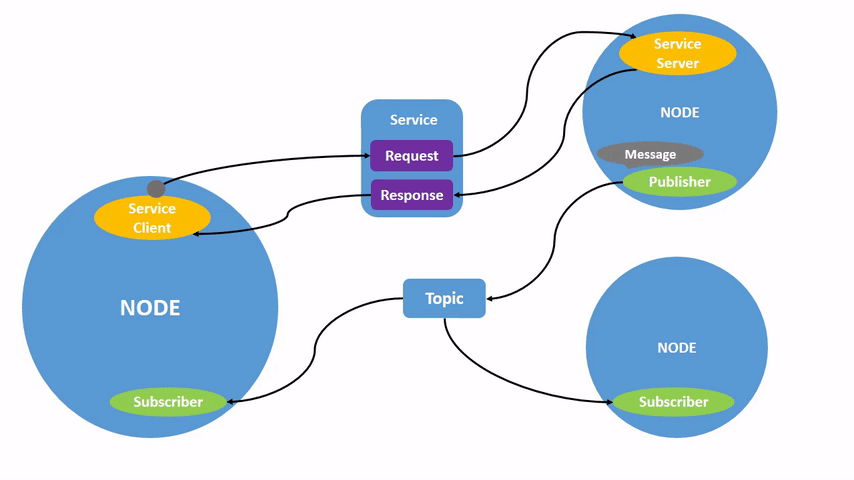


Figure 1: Nodes - Topic and Service [4]

### 2.1.2. Topics

Topics are named buses over which nodes exchange messages [5]. Topics have an anonymous publish/subscribe model, which decouples the assembly of data from its consumption. Nodes do not care who they are communicating with. Alternately, nodes who want to receive data subscribe to the relevant topic; nodes who want to send data, publish to the relevant topic. Topics are undirectional and streaming communication. That means a node can publish data to any number of topics and simultaneously subscribe to any number of topics. Figure 2 shows that topics can be one-to-many, many-to-one, and many-to-many communication. Each topic has a type with respect to ROS 2 message type used to publish to it and nodes can only get the messages with the matching type.

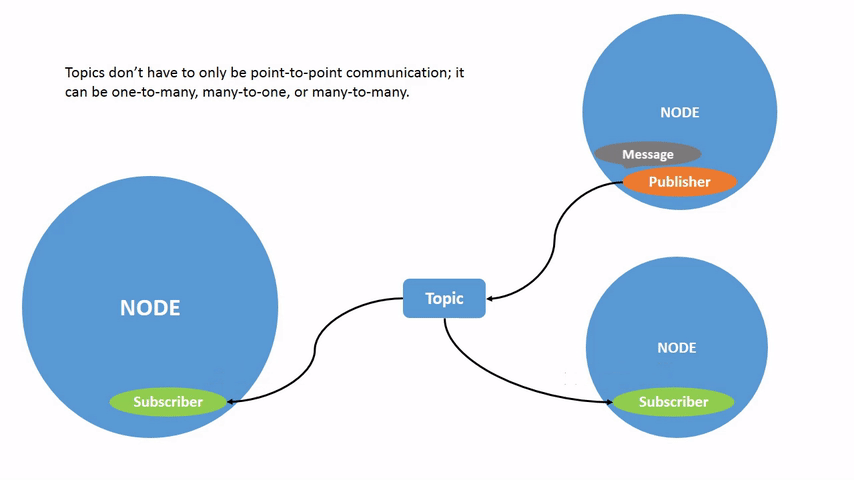


Figure 2: Topic - Multiple Publisher and Multiple Subscriber [5]

### 2.1.3. Parameters

A parameter is a configuration value of a node [6]. Parameters are the node settings. They can be stored in a node as integers, floats, booleans, strings, and lists. Each node can maintain its parameters in ROS 2. All parameters are dynamically reconfigurable via command lines or launch files. As it was the case in ROS 1, ROS 2 nodes allow configuration via command-line arguments to a certain degree [7]. Figure 3 shows how to use the command line to configure parameters. For ROS 2 specific command-line flags, it is used --ros-args and (--) is for user-defined ones.



Figure 3: ROS 2 specific arguments in command line [7]

Figure 4 shows how to configure ROS 2 specific arguments from the command line. The parameter burger\_mode is set to True.



Figure 4: ROS 2 configures argument using command line [8]

### 2.1.4. Launch files

Running several nodes in several terminals in a complex system is tedious and hard to control. The launch files give the ability to run multiple nodes simultaneously. The launch system in ROS 2 is responsible for helping the user describe the configuration of their system and then execute it as described [9]. What programs to run, where to run them, what arguments to pass are the configuration of the launch system. These reuse components, nodes throughout the system by giving them different configurations.

In ROS 2, writing launch files in Python is preferred. Launch files written in Python can start and stop different nodes as well as trigger and act on various events [9]. Figure 5 describes a launch file that has three nodes in the same package: turtlesim. The first two nodes have only one difference which is their namespace. Unique namespaces allow the system to start two simulators without node name nor topic name conflicts [10]. The final node has a different executable: mimic. This node has the remapping configuration.

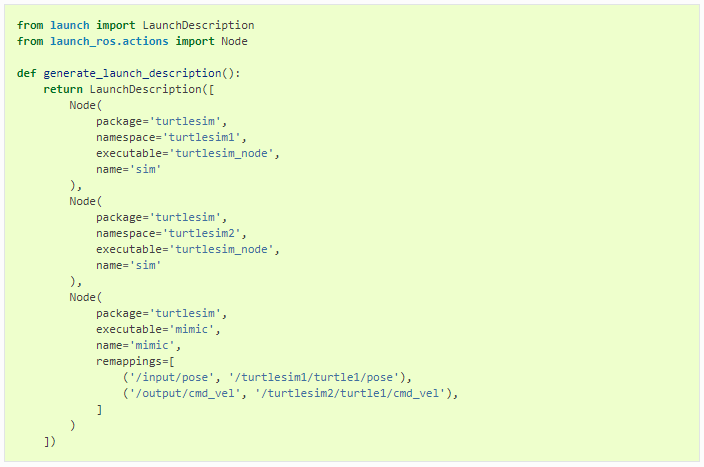


Figure 5: Launch file example [10]

Figure 6 shows how to run a launch file. The package name and launch file name are required to launch it.



Figure 6: How to run launch file [10]

### 2.1.5. Packages

Software in ROS is formed in packages. A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module [11]. The purpose of these packages is to give useful and common functionality so that they can be easily reused. ROS packages are enough capabilities to be useful, but not too much that the package is large and hard to use from other software. A package is a container of ROS code that can be built and released for others' use.

Package creation in ROS 2 uses ament as its build system and colcon as its build tool [12]. A package can be created using either Cmake or Python, which are officially supported, though other build types do exist.

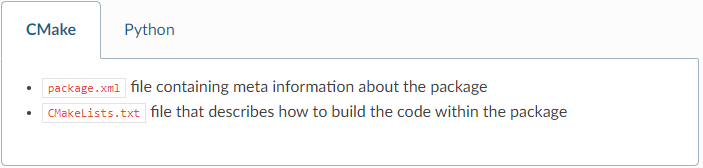


Figure 7: ROS 2 CMake packages with the minimum required contents [12]

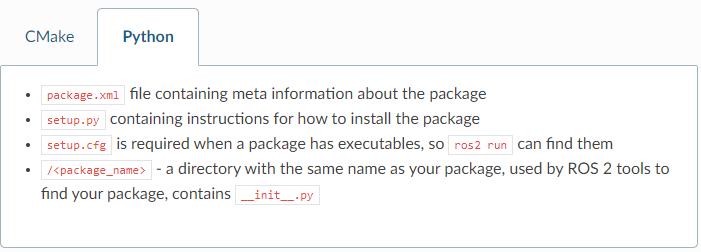


Figure 8: ROS 2 Python packages with the minimum required contents [12]

Figures 7 and 8 show the minimum required contents for ROS 2 CMake and Python packages. Furthermore, a package may have other directories which are shown in Figure 9.

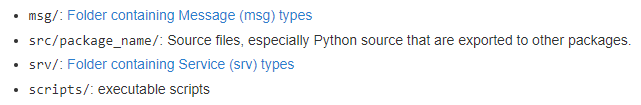


Figure 9: Other directories in ROS 2 packages [11]

### 2.1.6. Workspace

A workspace is a directory that contains ROS 2 packages. Packages need to be inside a workspace so that they can be built and run. Each workspace needs to be sourced in the terminal before using, which makes it available to run in that terminal.

There is an option to source “overlay” – a secondary workspace where we can add new packages without interfering with the existing ROS 2 workspace that we are extending. It calls “underlay”. The underlay must have all dependencies of all the packages in the overlay. Packages in the overlay will override packages in the underlay. Several layers of underlays and overlays are possible.

A single workspace can consist of as many packages as desire, each in their own folder. It is possible to have packages of different build types in one workspace (CMake, Python, etc.). Nested packages are not allowed. The best practice is to have a src folder in the workspace and then create packages in there, which make the top level of the workspace clean. Figure 10 displays how a typical workspace looks like in ROS 2.



Figure 10: A typical workspace in ROS 2

## 2.2. RTPS/DDS Interface: PX4-Fast RTPS(DDS) Bridge

The PX4-Fast RTPS(DDS) Bridge, which is also called the microRTPS Bridge, adds an RTPS interface to the PX4 Autopilot, enabling the transfer of Ubiquitous Object Request Broker (uORB) messages between the various PX4 Autopilot internal units and Fast DDS applications in real-time. It helps to better integrate with applications running and linked in DDS domains (including ROS nodes), making it easy to share commands, sensor data, location, and other vehicle information.

RTPS is the elemental protocol of the Object Management Group's DDS standard. It aims to enable scalable, real-time, dependable, high-performance, and interoperable data communication using the publish/subscribe pattern [13].

RTPS should be used when it is required to have reliably share time-critical/real-time information between the flight controller and offboard components. It is involved in cases where offboard software needs to become a peer of software components running in PX4 (sending and receiving uORB messages). One of the common use cases which are to get data to/from actuators and sensors where real-time is vital for vehicle control.

This section will explain the RTPS/DDS bridge architecture to get a better understanding of this interface.

### 2.2.1. Architectural overview

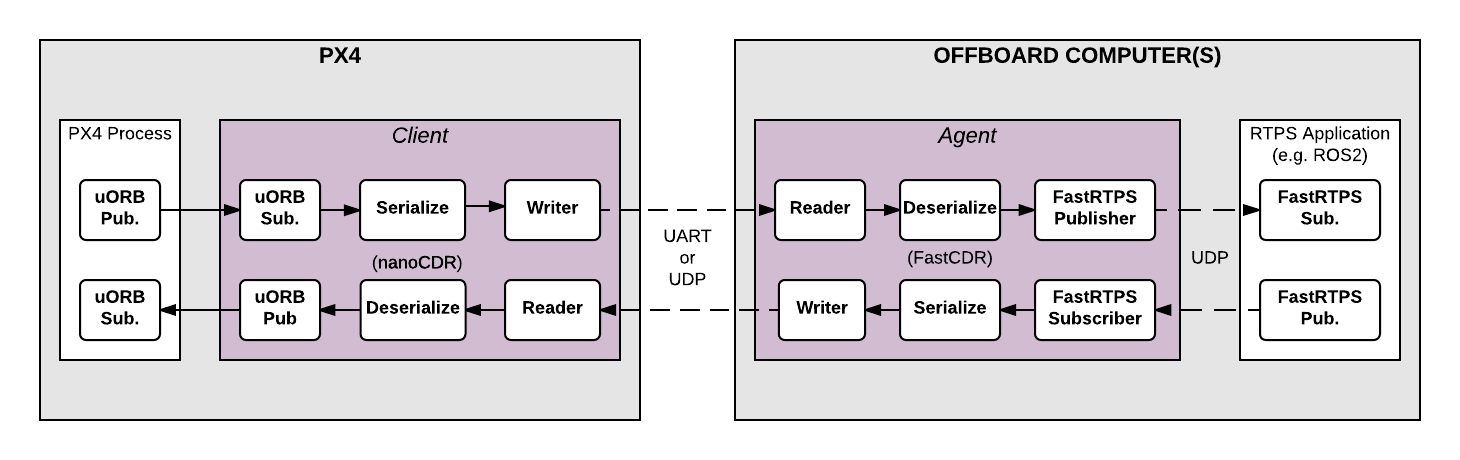


Figure 11: microRTPS Bridge Architecture

The microRTPS bridge exchanges messages between PX4 and DDS-participant applications, seamlessly converting between the uORB and RTPS/DDS messages used by each system [13]. The key components of the architecture are the client on the PX4 and the agent on the offboard computer which shows in the Firgure 11.

The microRTPS Client is the PX4 middleware daemon process that operates on the flight controller. It subscribes to uORB Publishers by other PX4 Process and then publishes messages to the Agent via Universal asynchronous receiver-transmitter (UART) or UDP; and also receives messages from the Agent and publishes them as uORB topics to the PX4 Autopilot.

The microRTPS Agent is a daemon process on an offboard computer (outside the flight controller), which can install ROS. The agent subscribes uORB messages from the Client and publishes them to RTPS applications, for example, ROS 2; and also subscribes messages from RTPS applications and forwards them to the Client.

The Client and Agent are linked via UART or UDP network. Common Data Representation (CDR) serializes the uORB messages before sending, which helps to create the same format of messages between different platforms. The Agent and RTPS applications are usually on the same device, connected to the Client through a wireless network or USB.

### 2.2.2. Code generation

Fast DDS 2.0.0 or later and Fast-RTPS-Gen 1.0.4 (not later) are required to generate the required code. When the PX4-Autopilot is compiled, all the code required to create, build and run the bridge is automatically generated.

### 2.2.3. uORB messages support

The generated bridge code will enable a specified subset of uORB topics to be published/subscribed via RTPS, regardless if you are deploying a ROS application or not [13]. The file called ***uorb\_rtps\_message\_ids.yaml*** in the ***PX4-Autopilot/msg/tools/*** directory defines the set of uORB messages to be used with RTPS for code generation, which is to be sent, received, or both, and the RTPS ID for the message to be used in DDS middleware. Each RTPS message requires an ID to be set in the file, shown in Figure 12.

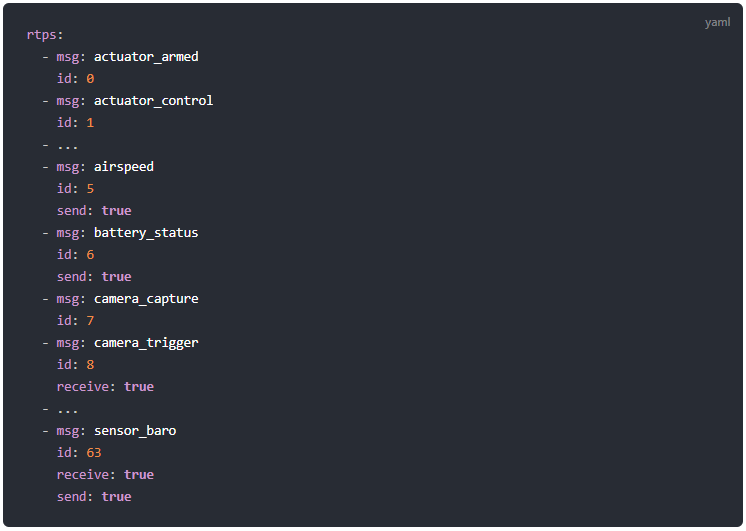


Figure 12: uorb\_rtps\_message\_ids.yaml file [13]

### 2.2.4. Client (PX4/PX4-Autopilot)

The source code of the Client is generated, compiled, and built into the PX4 Autopilot firmware as part of the normal build process. To build the PX4 Autopilot firmware for NuttX/Pixhawk flight controllers use the \_rtps feature, for example, a SITL target shown in Figure 13 below.



Figure 13: Build PX4 SITL RTPS [13]

The SITL device uses Gazebo which is a powerful 3D simulation environment for autonomous robots. A well-designed simulator makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI systems using realistic scenarios [14]. Gazebo offers robot simulation in complex indoor and outdoor environments. Most of all, Gazebo is free with a huge active community.

The Client can be launched from NuttShell/System Console. The command syntax is shown in Figure 14 below:

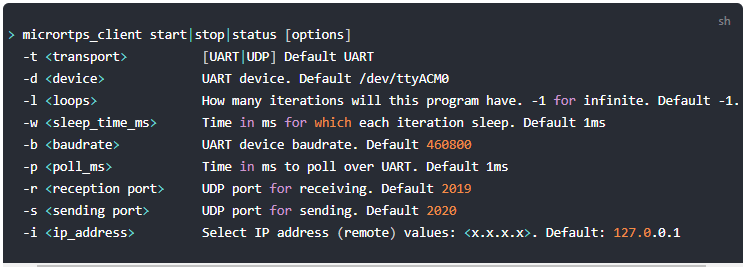


Figure 14: Launching Client options [13]

For instance, to run the Client daemon with SITL connecting to the Agent via UDP is shown in Figure 15.



Figure 15: Starting the Client using UDP [13]

### 2.2.5. Agent in an Offboard Fast DDS interface (ROS-independent)

The Agent code is automatically generated when the associated PX4 Autopilot firmware is built. The command syntax for the Agent is listed in Figure 16 below.

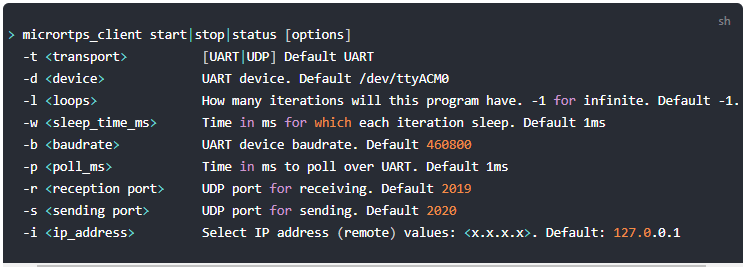


Figure 16: The Agent command options [13]

For example, to start the micrortps\_agent with a connection through UDP is shown in Figure 17 below:



Figure 17: Starting the Agent using UDP [13]

## 2.3. Quality of Service Settings

ROS 2 provides a collection of QoS policies that allow developers to adjust communication between nodes. ROS 2 can be as reliable as TCP or as best-effort as UDP depending on the settings of the QoS policies. Dissimilar to ROS 1, which uses TCP as the primary transport, ROS 2 gains the flexibility of the underlying DDS transport where the best-effort policy is more suitable in environments of lossy wireless networks and ROS 2 can meet the deadlines in real-time systems with the right set of QoS profile.

QoS profile is formed by a set of QoS policies. ROS 2 offers a set of predefined QoS profiles for common use cases, for instance, publishers or sensor data. And developers can be able to change the default QoS policies for their will. QoS profiles can be defined for publishers, subscriptions, service servers, and clients. Each of them can have a specific profile, but in order to communicate, they must be compatible.

This chapter will give the basics of QoS Settings in ROS 2 including their policies, compatibilities, events and, comparison to ROS 1.

### 2.3.1. QoS policies

The main QoS profiles have setting options which are shown in Table 1 below. Each policy has the default option depending on the use cases of publishers, subscriptions, services, etc...

|  |  |  |
| --- | --- | --- |
| QoS policies | Setting options | Explanation |
| **History** | *Keep last* | Store up to the last N samples, configurable via the depth option |
|  | *Keep all* | Store all samples |
| **Depth** | *Queue size* | Only effective when the “history” policy was set to “keep last” |
| **Reliability** | *Best effort* | Attempt to deliver samples and may lose some when the network is lossy |
|  | *Reliable* | Guarantee all the samples are delivered and may try several times |
| **Durability** | *Transient local* | The publisher turns into responsible for persisting samples for “late-joining” subscriptions. |
|  | *Volatile* | No aim to persist the samples |
| **Deadline** | *Duration* | The expected maximum amount of time between subsequent messages being published to a topic [15] |
| **Lifespan** | *Duration* | The maximum amount of time between the publishing and the reception of a message without the message being considered stale or expired (expired messages are silently dropped and are effectively never received) [15]. |
| **Liveliness** | *Automatic* | The system will consider all of the node’s publishers to be alive for another “lease duration” when any one of its publishers has published a message [15]. |
|  | *Manual by topic* | The system will consider the publisher to be alive for another “lease duration” if it manually asserts that it is still alive (via a call to the publisher API) [15]. |
| **Lease Duration** | *Duration* | The maximum period of time a publisher has to indicate that it is alive before the system considers it to have lost liveliness (losing liveliness could be an indication of a failure) [15]. |

Table 1: QoS policy setting options

The “history” and “depth” policies in ROS 2 together have functionality similar to the queue size in ROS 1.

The “reliability” policy in ROS 2 is similar to either UDPROS (only in roscpp) for BEST\_EFFORT or TCPROS (ROS 1 default) for RELIABLE in ROS 1. The RELIABLE policy in ROS 2 is even implemented using UDP, which allows for multicasting if needed.

The “durability” policy TRANSIENT\_LOCAL, combined with any depth, provides functionality similar to that of “latching” publishers [15]. The remaining policies in ROS 2 are not similar to any in ROS 1 which means ROS 2 is more powerful and featureful than ROS 1. The core team of ROS 2 promises that there will be even more QoS profiles in the future.

### 2.3.2. QoS profiles

ROS 2 offers a range of defined QoS profiles which works well in common use cases, that allows developers to focus on other aspects instead. There are several defined QoS profiles, which are for publishers, subscriptions, services, sensor data, parameters, and system default.

For publishers and subscriptions, it is important to deliver all the messages so by default, reliability is set to RELIABLE, history is set to KEEP\_LAST with a queue size of 10, durability is set to VOLATILE, liveliness is set to DEFAULT. Deadline, lifespan, and lease durations are also all set to DEFAULT.

Services are also similar to publishers and subscriptions. Therefore it is reliable. The main difference between them is that services have to use VOLATILE durability. Otherwise, re-started service servers may receive outdated requests.

For sensor data, it is important to get the latest samples, rather than receiving all of them and may lose some of the samples. Therefore, the sensor data profile uses BEST\_EFFORT reliability and a smaller queue size.

Parameters have the same profile as services. The main difference is that a much larger queue depth than services is required in parameters in order to prevent requests get lost when the parameter client is not able to reach the parameter service server.

### 2.3.3. QoS compatibilities

A connection between a publisher and a subscription or a service and a client is only made if the pair has compatible QoS profiles. QoS profile compatibility is determined based on a “Request vs Offered” model [15]. The requested QoS profile should not be more stringent than that of offered QoS profile. Tables 2 shows the examples for compatibilities of reliability and durability.

|  |  |  |  |
| --- | --- | --- | --- |
| QoS policies | Publisher | Subscription | Compatible |
| **Reliability** | *Best effort* | *Best effort* | Yes |
|  | *Best effort* | *Reliable* | No |
|  | *Reliable* | *Best effort* | Yes |
|  | *Reliable* | *Reliable* | Yes |
| **Durability** | *Volatile* | *Volatile* | Yes |
|  | *Volatile* | *Transient local* | No |
|  | *Transient local* | *Volatile* | Yes |
|  | *Transient local* | *Transient local* | Yes |

Table 2: QoS compatibilities of reliability and durability

### 2.3.4. QoS events

There are some possible events related to QoS policies. If developers want to handle these events, they can create callback functions for each publisher or each subscription similar to how to handle the topic‘s messages. Table 3 lists all the possible QoS events in ROS 2.

|  |  |  |
| --- | --- | --- |
| Node | QoS events | Explanation |
| **Publisher** | *Offered deadline missed* | The publisher is not able to publish a message within the duration that was set out by the deadline QoS policy. |
|  | *Liveliness lost* | The publisher has failed to indicate its liveliness within the lease duration [15]. |
|  | *Offered incompatible QoS* | The publisher has found out a subscription on the same topic is requesting a QoS profile that it can not satisfy. |
| **Subscription** | *Requested deadline missed* | The subscription is not able to receive a message within the duration that was set out by the deadline QoS policy. |
|  | *Liveliness changed* | The subscription has noticed that one or more publishers on the subscribed topic have failed to indicate their liveliness within the lease duration [15]. |
|  | *Requested incompatible QoS* | The subscription has found out a publisher on the same topic is offering a QoS profile that it can not be satisfied with. |

Table 3: QoS events

## 2.3. Evaluation tools

### 2.3.1. Linux network traffic control utility

Linux network traffic control (tc) is a useful utility that can configure the kernel packet scheduler. In this thesis, packet loss simulation is required in order to evaluate the QoS feature in ROS 2. With this traffic control, packet loss simulation can easily achieve with one command. To use *tc, iproute* must be installed. Figure 18 shows how to install *iproute* in Linux.



Figure 18: Installing iproute for tc

Figure 19 describes how to simulate 10% of packet loss using the scheduler qdisc to adda new rule to device dev wlp2s0on outbound traffic scheduler rootwith the network emulator netem.



Figure 19: Simulating 10% packet loss

### 2.3.2. ROS 2 Topic Statistics

Like ROS 1, ROS 2 also provides the measurement of statistics for messages received in the subscription nodes. This Topic Statistics helps developers to evaluate the performance of the overall systems and diagnose any current issues.

The Topic Statistics can measure the received message age and the received message period. The average, maximum, minimum, standard deviation, sample count are calculated for each measurement in the moving windows. The received message period and the received message age are both calculated in milliseconds. While the former uses the system clock to measure the period between received messages, the latter requires the timestamp in the header field of the message to calculate the age of messages.

The Topic Statistics is not enabled by default. Therefore, developers have to enable on their own if they want to get the statistics. The data is published on the default topic /statistics at the default period of 1 second. Developers otherwise can reconfigure the default topic name and the period as well. If the messages received do not have a timestamp in the header, all statistics values in the received message age are Not a Number (NaN). The first sample of the received message period will not have a measurement because the previous message that arrived is required to calculate the statistics.

## 2.4. Related works

Since the release of the alpha version, ROS 2 was evaluated by [16] who set up several communication cases between nodes in ROS 1, ROS 2, and ROS 1 and ROS 2 using ROS1-bridge. The experiments were implemented in which nodes are in the same machine and two separate machines using wireless networks as well in all three DDS implementations including Context, OpenSlice and, FastRTPS. Messages with sizes between 256 B and 4 MB were used to transfer at a 10 Hz rate. QoS policies had also been applied with RELIABLEand BEST\_EFFORT. They evaluated capability, latencies, throughput, the number of threads, and memory consumption. They believe ROS 2 with DDS supports is superior to ROS 1 because it can save past data with QoS policies and does not require a master node.

Latency in ROS 2 Multi-Node Systems had been analyzed by [17] in this year – 2021. They set up the experiments with variable parameter values including a range of publisher frequency: 1, 10, ..., 90, 100; payload with 128 B, 1 KB, 10 KB, 100 KB, 500 KB; 3, 5, ..., 21 ,23 nodes; Context, FastRTPS, CycloneDDS; and RELIABLE***,*** BEST\_EFFORT reliability. They concluded some of the rules: the higher the frequency, the lower the latency; payload size higher than fragmentation size of UDP (64 KB), latency increases with the payload size; latency strongly depends on the hardware used and the parameter setting and the middleware do not yield the lowest latency in all cases. One of their conclusion that is alike to this thesis results which is using RELIABLE transmission imposed a latency overhead of up to 15% compared to BEST\_EFFORT.

ROS 2 had been also evaluated in lossy unmanned networks by [18] in 2020. They evaluated the network performance of ROS 2 with different QoS profiles and security settings. Latency and message loss rates are examined. The experiments used network simulation NS-3 to simulate lossy wireless networks between ROS 2 nodes. They found that security and the number of nodes had a strong impact on all QoS profiles regarding message latency and message drop rate. They also found that the Sensor profile which has BEST\_EFFORT reliability outperformed the Default and Parameter profiles over all simulations, which have RELIABLE reliability.

# 3. Experimental Design and Setup

In this chapter, we describe the experiment’s architecture used to evaluate QoS policies in ROS 2. The implementation of the experiments has also been explained using both DEFAULT and BEST\_EFFORT policy of reliability.

## 3.1. Architecture of the experiments

### 3.1.1. Unmanned aerial vehicles in fire fighting

In this section, the application of UAVs in fire fighting will be described. This thesis aims to support these UAVs in this context. In the ongoing research, in the use case of fire fighting, UAV swarm is working together to help firefighters find victims and extinguish the fire. The swarm can work in manual mode and autonomous mode. There is a Ground Control Station (GCS), who will send orders to the UAV swarm. In manual mode, the GCS only sends the trajectory of setpoints to the drone. In the autonomous mode (which should be used primarily), the GCS mostly sends high-level missions, for example, "explore this volume" or "return to base" to the Companion Computer (Figure 20), which then plans the trajectory and sends to the PX4 over the stable serial connection. It may also send other information such as the status of other drones. Therefore, the relationship is one-to-many (one GCS – many drones). In this thesis, we only consider the one-to-one relationship (one GCS – one drone) and GCS only sends Setpoint (Figure 20) to the Companion Computer. Nevertheless, it is one of several use cases in reality.

### 3.1.2. Architecture

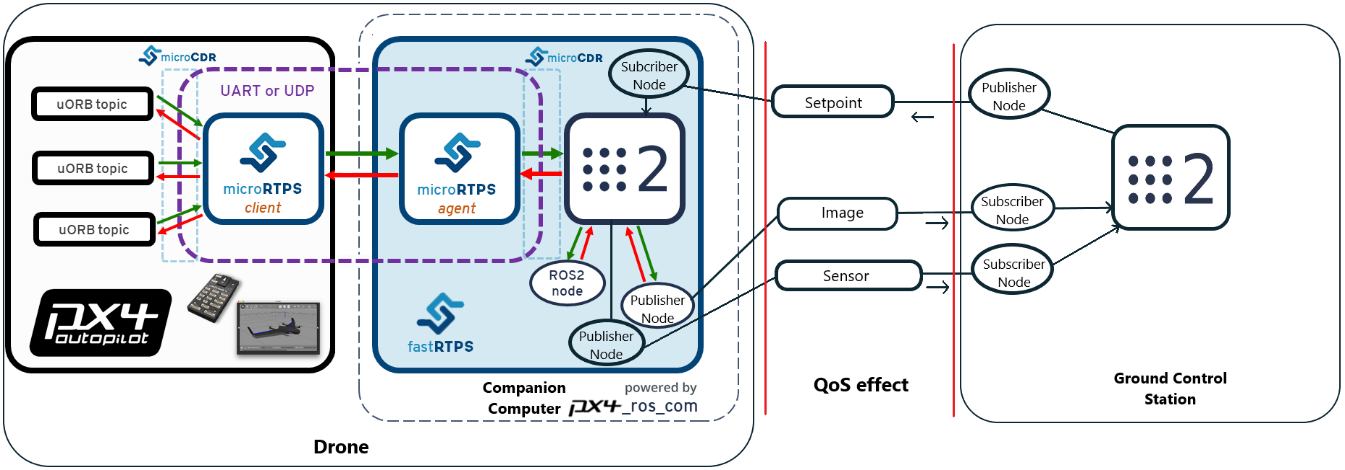


Figure 20: The architecture of the experiments

Figure 20 shows the architecture of the experiments overall. There are one Ground Control Station and one Drone connecting via a wireless network. The GCS and the Companion Computer have ROS 2 installed. They are communicating Setpoints, HD images and, Sensor data. While the GCS publishes Setpoints to the Companion Computer, the Companion Computer publishes Images and Sensor data getting from the PX4 to the GCS. Therefore, the GCS has one publisher node and two subscriber nodes, the Companion Computer has two publisher nodes and one subscriber node. In the Drone, the Companion Computer connects to PX4 via a stable serial connection which is not affected by lossy wireless networks. The microRTPS client on the PX4 subscribes uORB topics (for example, here, Sensor data from the drone) and then publishes them to the microRTPS agent on the Companion Computer, and also receives messages (here, Setpoints) from the microRTPS agent and then publishes them as uORB topics to the PX4 Autopilot. On the Companion Computer, there is a microRTPS agent which subscribes messages from DDS participant of ROS 2 installed and then publishes them to microRTPS client as uORB topics, and also receives messages from microRTPS client, and then publishes them to the DDS participant. The connection between the Companion Computer and the GCS is established by wireless networks which are usually lossy and unstable. Consequently, QoS settings influence this connection. Therefore, Setpoints, Images and, Sensor data messages are affected. This thesis is evaluating message ages and message period of Setpoints, Images and, Sensor data in both RELIABLE and BEST\_EFFORT reliability QoS policy.

### 3.1.3. Package list in the workspace

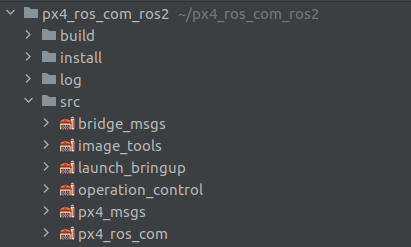


Figure 21: List of packages in the workspace

In the experiments, there are six packages in the workspace which are shown in Figure 21. These packages will be explained in detail below:

**bridge\_msgs:** defines messages to be communicated between the Companion Computer and the GCS, for instance, **Sensor.msg** (Figure 22)includes combined sensor data, **Setpoint.msg** (Figure 22)includes coordinate x, y, z. These files also contain a *Header* that has ***frame\_id*** and ***stamp*** information.

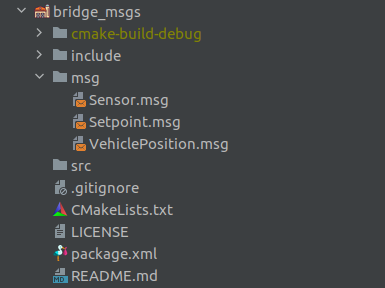
****

Figure 22: bridge\_msgs package structure

**image\_tools:** is forked from ROS2 foxy source code [19] and then modify to adapt the experiments. Figure 23 shows the structure of the image\_tools package. In the experiments, **cam2image.cpp** and **showimage.cpp** are used to transfer images from the Companion Computer to the GCS.

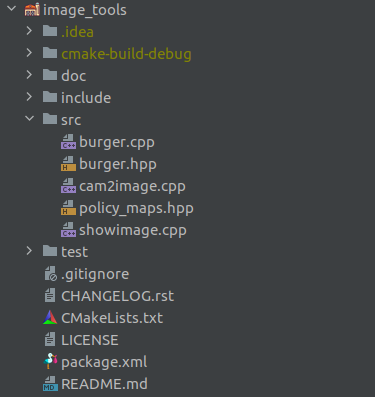
****

Figure 23: image\_tools package structure

**launch\_bringup:** contains launch files of the workspaces which help to start multiple nodes in only one command. In the *launch* folder, four different launches have two options: *default* and *best\_effort* QoS policy for both drone and the GCS.

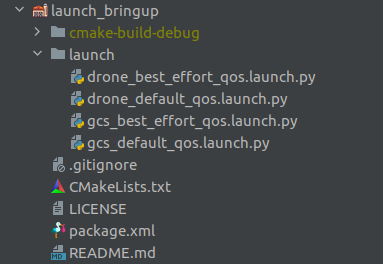
****

Figure 24: launch\_bringup package structure

**operation\_control:** is the main package of the GCS which contains an advertiser that sends Setpoint to the Companion Computer and a listener that subscribes sensor data from the Companion Computer. Figure 25 describes the structure of the package.

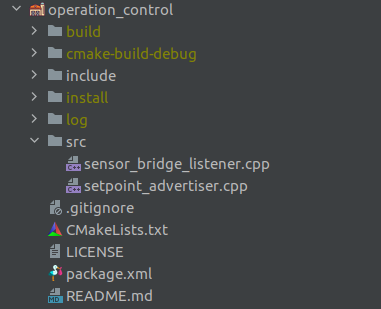
****

Figure 25: operation\_control package structure

**px4\_msgs** (Figure 26)**:** is forked from ROS2 message definitions of the PX4 Pro ecosystem [20]. After building the package, it will automatically generate all required interfaces to match with ROS 2 nodes with the PX4 Autopilot internal. This package has a strong dependency on **px4\_ros\_com** package.

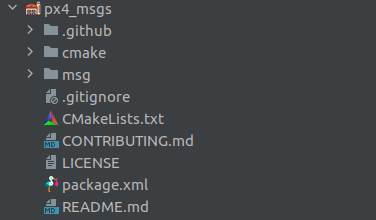
****

Figure 26: px4\_msgs package structure

**px4\_ros\_com** (Figure 27)**:** is forked from PX4-ROS2 bridge [21] and then edit to adapt the experiments. While **offboard\_control.cpp** is used to make the drone take off and receive Setpoint from the GCS to move the drone, the **sensor\_combined\_listener.cpp** is used to get sensor data from the drone and then publish them to the GCS.

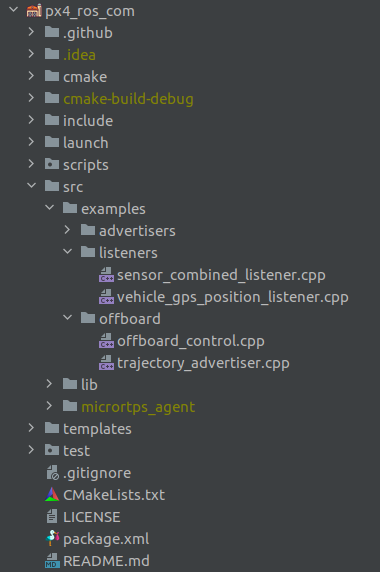
****

Figure 27: px4\_ros\_com package structure

### 3.1.4. System setup

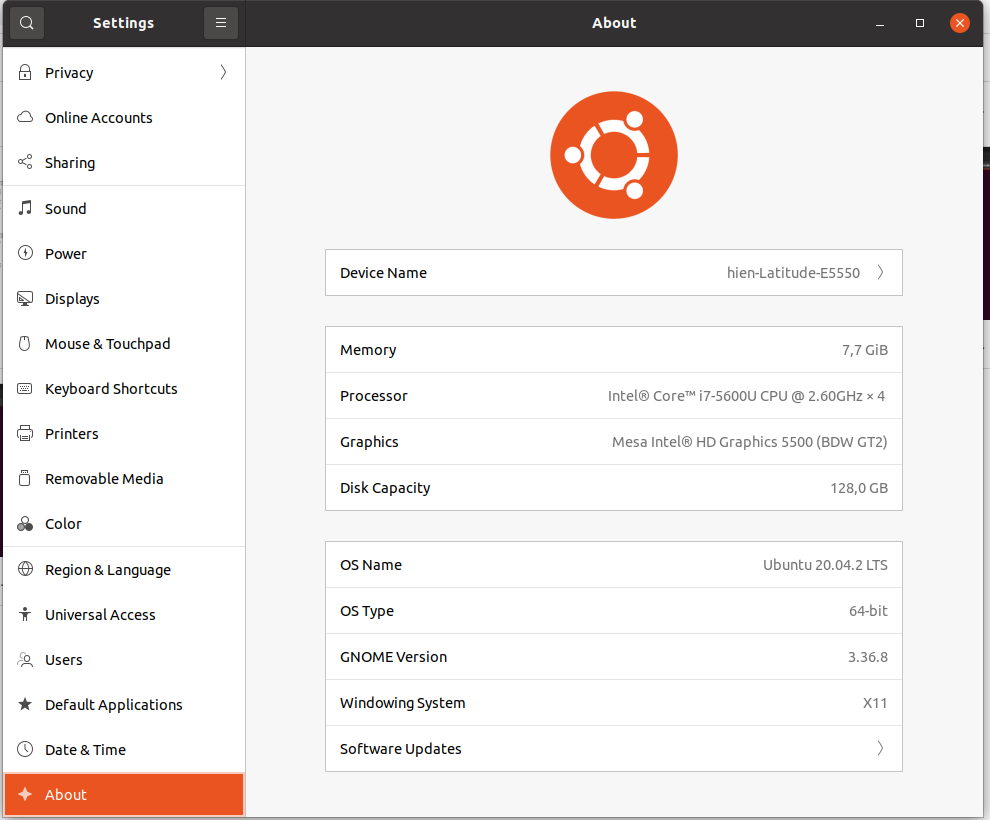


Figure 28: System specification

The experiments use two PCs (one for the Companion Computer and one for the GCS) that have the same system specification as below (Figure 28 shows the details of the system):

* Processor: Intel® Core™ i7-5600U CPU @ 2.60GHz × 4
* Memory: 8 GB
* Disk Capacity: 128,0 GB SSD
* OS: Ubuntu 20.04.2 LTS
* ROS 2 version: ROS 2 Foxy Fitzroy - Patch Release 4

## 3.2. Experiment with the default policy of QoS settings

### 3.2.1. Introduction

In this section, the experiment uses **RELIABLE** for RELIABILITY policy. The default policy of RELIABILITY in ROS 2 is **RELIABLE**. In this policy, publishers will try to send the messages multiple times until they get the acknowledgment from subscribers. Linux network traffic control (tc) is used to simulate packet loss from 0% to 90%. Message ages and message periods are calculated by ROS 2 Topic Statistics. Message ages and message periods are primary measurement indicators. Furthermore, the loss messages’ number of Setpoint messages is also observed in terminals as for the drone, it is important to not lose any Setpoints which can lead to failing the mission.

### 3.2.2. Implementation

As mentioned before in the architecture section, there are one publisher node and two subscriber nodes in the GCS; and there are two publisher nodes and one subscriber node in the Companion Computer. All these nodes will be described briefly in this section.

**setpoint\_advertiser.cpp**: is the setpoint publisher in the **operation\_control** package which publishes Setpoints through a topic with a frequency. Figure 29 describes the implementation of the setpoint publisher. Firstly, the publisher is created using message type **Setpoint** from **bridge\_msgs** package which publishes the topic “Setpoint\_PubSubTopic” (line 28). Then creating a wall timer with the 100 ms (10 Hz) period and a call back (line 44). In the timer callback, a Setpoint is created with **x**, **y**, **z**. While **x** is increasing by counting, **y** and **z** remain unchanged. The **stamp** and **frame\_id** are also put into the Setpoint’s header for statistics. Finally, the Setpoint is published on the topic defined above (line 42).

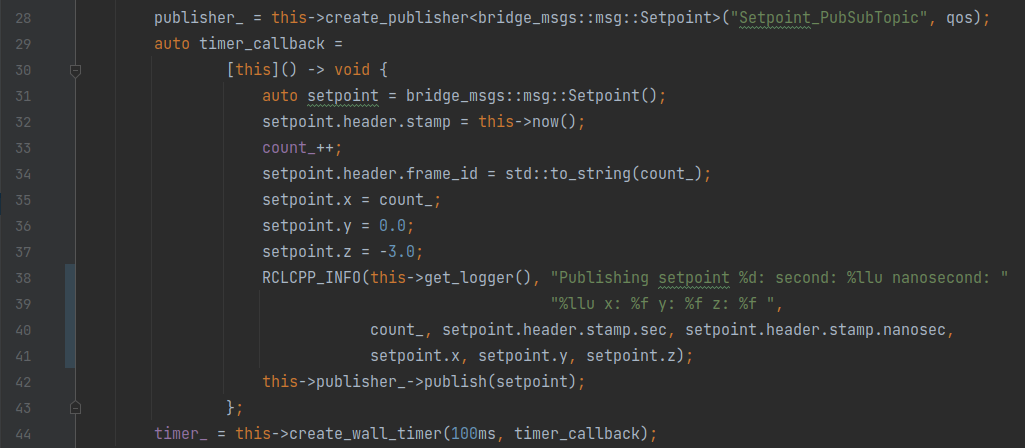


Figure 29: Advertiser is publishing Setpoints

**offboard\_control.cpp** [21]: is written by PX4 team and the community to control the drone, making it take off and move. We have added more code to make it to be the setpoint subscriber in the **px4\_ros\_com** package which subscribes “Setpoint\_PubSubTopic” to receive Setpoints from **setpoint\_advertiser.cpp** node. Figure 30 shows how it works. First, the subscription is created that subscribes to the topic “Setpoint\_PubSubTopic” with a **setpoint\_callback**, the **options,** and a **qos** parameter (line 127). The **options** and **qos** will be explained in the second experiment with the BEST\_EFFORT policy. In the **setpoint\_callback**, the **Setpoint** is received from the publisher, and with that information, a **TrajectorySetpoint** (a message from **px4\_msgs** package which is compatible with PX4 devices) is created with **x**, **y**, **z** from the **Setpoint**. The reason why we do not use **TracjectorySetpoint** directly is that it does not contain a header with a stamp which is the requirement for calculating message age for statistics. The **TracjectorySetpoint** is then published by **trajectory\_setpoint\_publisher** (is created before to make the drone enter the offboard mode) to the PX4 and then make the drone moving (line 124).

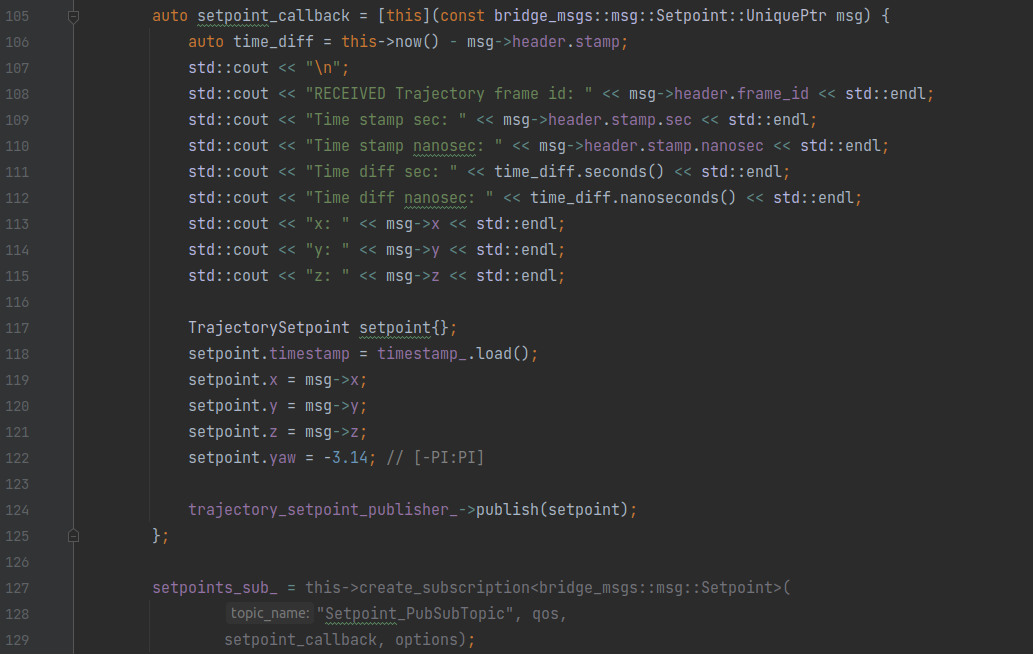


Figure 30: Subscribing Setpoint topic

**sensor\_combined\_listener.cpp** [21]: is created by the PX4 team and the community to subscribe sensor data from the drone in **px4\_ros\_com** package. Here we add more code to publish the sensor data so that other nodes can subscribe and get the data. In Figure 31, a subscription is created subscribing to the topic “SensorCombined\_PubSubTopic” using the message type **SensorCombined** from **px4\_msgs** package with a **sensor\_combined\_callback** (line 95). Inside the **sensor\_combined\_callback** implementation (line 66), the **SensorCombined** message is received and printed out all parameters of the message. Then a **Sensor** message from the **bridge\_msgs** package is initialized and then set the message’s parameters by getting **SensorCombined’s** parameters. The reason why we do not use **SensorCombined** directly instead of creating the new **Sensor** message is the same as the reason above because **SensorCombined** does not have a header for the statistics. Then the created message is published the **sensor\_publisher** (line 93) which is initiated before (line 64) using **Sensor** message to publish the topic named “SensorBridge\_PubSubTopic”.

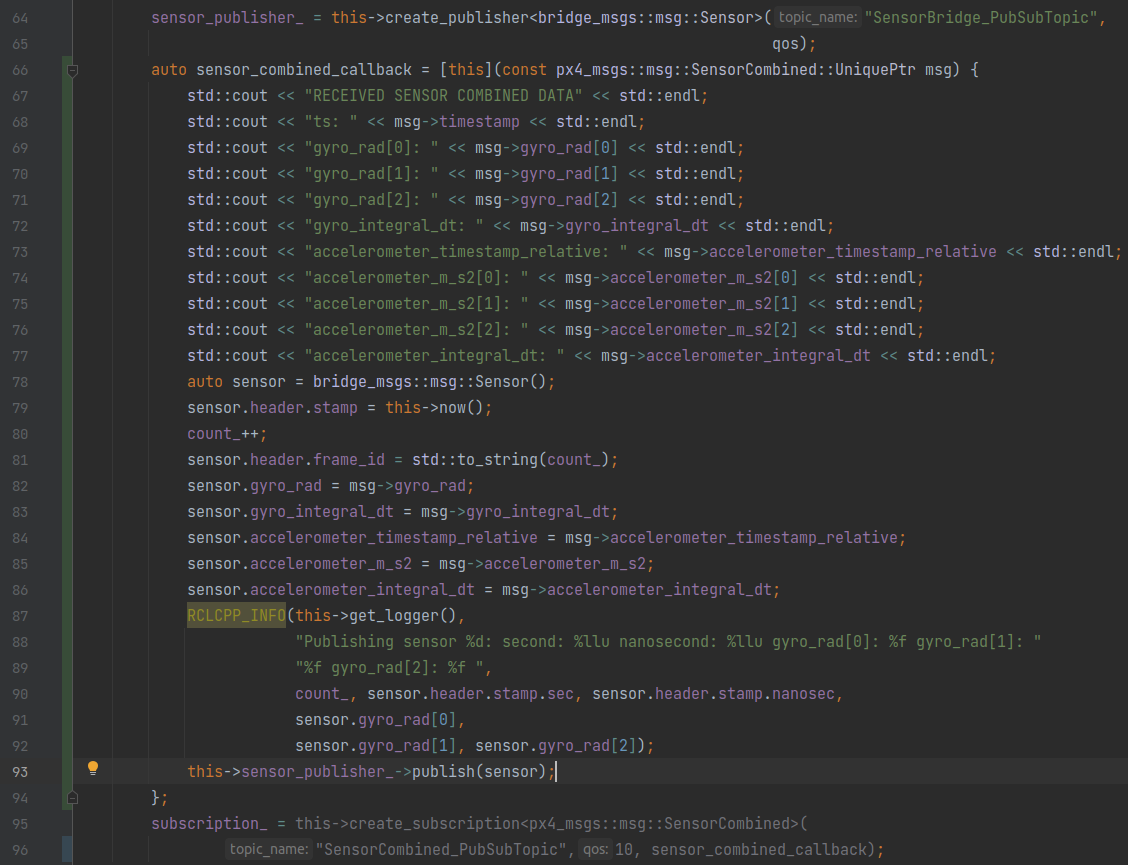


Figure 31: Publishing sensor data of the drone

**sensor\_bridge\_listener.cpp**: is the sensor subscriber in the **operation\_control** package which subscribes to the topic “SensorBridge\_PubSubTopic” using the type message of **Sensor** from **bridge\_msgs** package with the **sensor\_callback** (line 48 in Figure 32). Inside the implementation of **sensor\_callback**, all parameters of the message are printed for observation.



Figure 32: Subscribing Sensor data

**cam2image.cpp** [19]: is the image publisher which publishes ROS Image over the topic name “image” in the **image\_tools** package. Figure 33 shows the publisher is created using the **Image** message with topic name and **qos** which will be explained in the next experiment. In this node, source images are from the Companion Computer’s camera. The images will be converted from OpenCV type to ROS type before publishing. The default size of the image is 320 pixels of width and 240 pixels of height. For frequency of transmission, the default is 30 Hz.

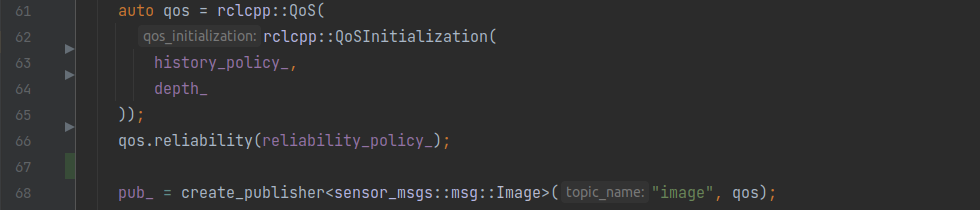


Figure 33: Image publisher [19]

**showimage.cpp** [19]: is the image subscriber which subscribes to the topic “image” using the **Image** message type with a callback and other parameters (Figure34). In the callback’s implementation, images are processed to show them on the GCS’s screen.

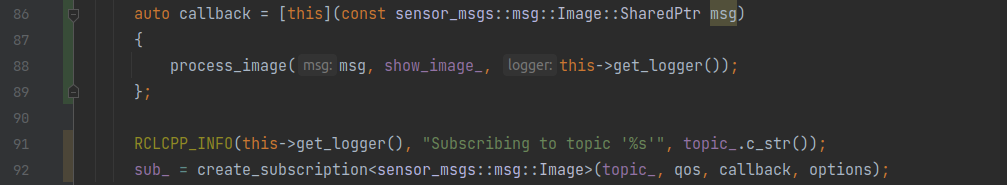


Figure 34: Image subscriber [19]

For the full implementation of the experiment for the DEFAULT policy as well as how to build and run it, please follow the link in this [22] reference.

## 3.3. Experiment with the best\_effort policy of QoS settings

### 3.3.1. Introduction

In this section, the experiment uses **BEST\_EFFORT** for RELIABILITY policy. In this policy, publishers will attempt to send the messages without requiring acknowledgment from subscribers and may lose them if the network is weak. Other tools are used as the same as the first experiment, for instance, Linux network traffic control (tc) is used to simulate packet loss from 0% to 90%; message ages and message periods are calculated by ROS 2 Topic Statistics. Message ages and message periods are the main measurement factors. Moreover, the loss messages’ number of Setpoint messages is also evaluated as it is vital to not lose any Setpoints which can lead to failing the mission. In the last experiment, the six publishers/subscriber's implementation has been explained. In this section, the **BEST\_EFFORT** experiment also uses these six nodes. Therefore, only the QoS settings implementation will be described in this section.

### 3.3.2. Implementation

To change the default policies of QoS, we must define which policies we want to use in the node. In this experiment, we define three policies that can be changed their policies at the node’s initiation. Figure 35 shows three policies: DEPTH, RELIABILITY, HISTORY. There are also two maps for picking the corresponding policies depending on the parameters that we set up for the nodes in the beginning. For RELIABILITY policy, there are “**reliable**” and “**best\_effort**”; for HISTORY policy, there are “**keep\_last**” and “**keep\_all**”. Note that in the scope of this thesis, only RELIABILITY policy is subject to change.

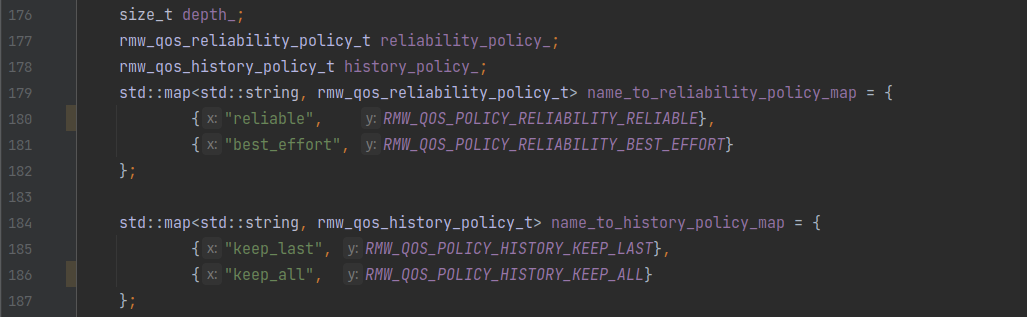


Figure 35:QoS policies definition [19]

Instead of starting one node in a terminal, we use launch files in ROS 2 which can run multiple nodes simultaneously. Figure 36 shows an example of a launch file that start 3 nodes: **showimage**, **sensor\_bridge\_listener**, and **setpoint\_adversiser** with the same “**best\_effort**” parameter for “**reliability**” policy. That means when this launch file is used, the default QoS policy of “**reliability**” is changed to “**best\_effort**”.



Figure 36: GCS best\_effort launch

After running the launch files, these parameters in the launch files need to be parsed to pick corresponding values. Figure 37 describes how the parameters are setting by the values that users have set at run-time. For instance, in line 204, **reliability\_param** is set by using “declare\_parameter()” function which uses the value that is set by users at run-time, otherwise, the default value (“**reliable**”) is set. Consequently, the compatible constant for QoS policy that ROS understands is found using the defined maps (from line 206 to line 212). For HISTORY and DEPTH policy, the processes are the same.

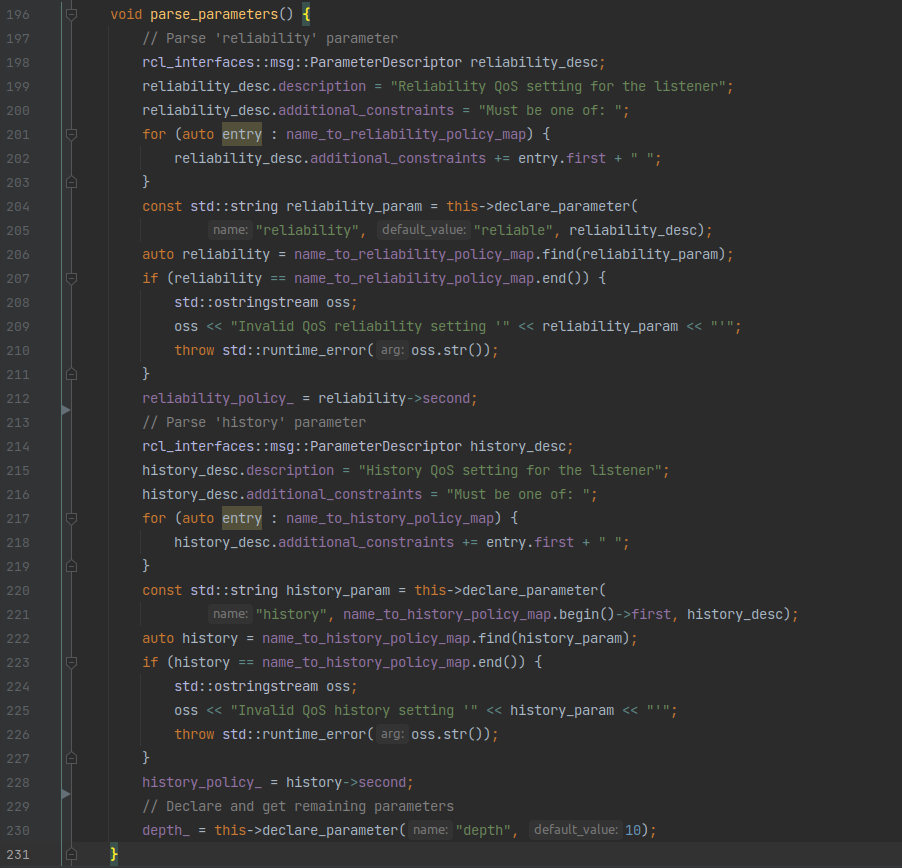


Figure 37: Parsing parameters [19]

After parse\_parameters(), a **qos** is initialized using these variables above (Figure 38).

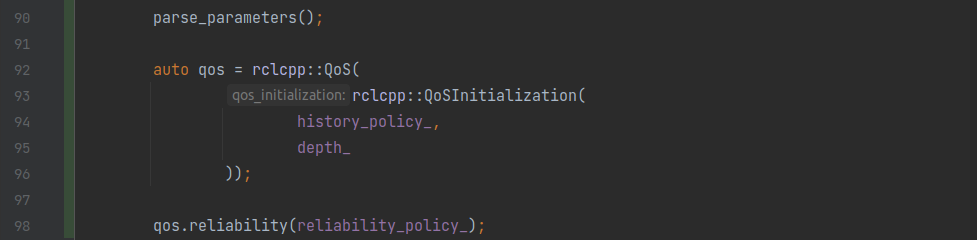


Figure 38: QoS Initialization [19]

When the nodes are running, how can we calculate the message ages and message periods? In this thesis, we use ROS 2 Topic Statistics. By default, the statistics function is not enabled so we need to manually enable it via options (lines 101 and 102 in Figure 39). These message ages and message periods are published to the topic named “/statistics” by default. However, in the experiments, three different subscribers should be tracked so we change the statistics topic name, for **Setpoint** data, we use “/statistics\_setpoint”; for **Image** data, we use “/statistics\_imgae”; for **Sensor**, we use “/statistics\_sensor”.

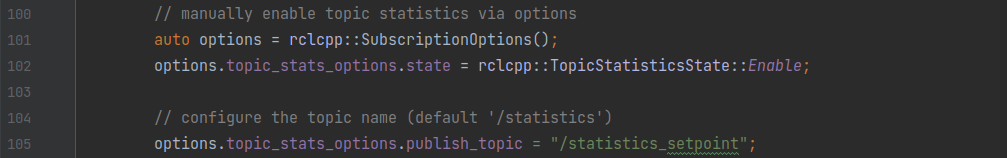


Figure 39: Topic Statistics enabling

For the source code of both experiments and how to build and run the BEST\_EFFORT experiment, please follow the link in this [22] reference.

# 4. Results Discussion

In this chapter, the results of the experiments will be discussed. The message ages and the message periods in different percent numbers of packet loss from 0% to 90% will be described. The last section will compare the corresponding results between these two experiments.

## 4.1. Experiment with the default policy of QoS settings

In this section, we will observe the results of the **RELIABLE** (default)of RELIABILITY policy. Firstly, the Setpoint statistics data is discussed, and then the Image statistics data, finally, the Sensor statistics data is explained.

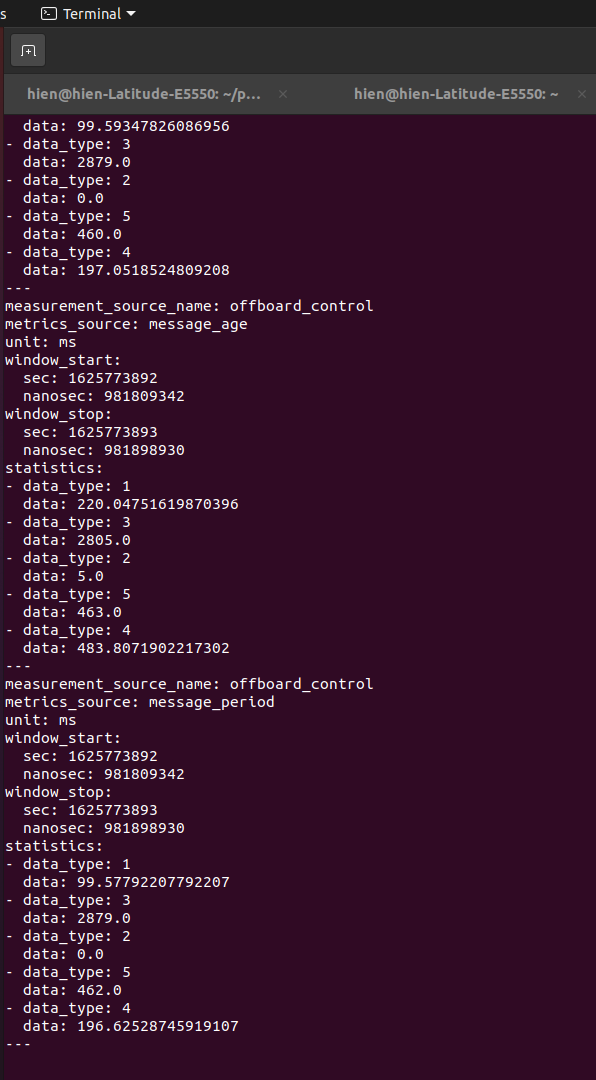


Figure 40: Reliable output terminal of topic /statistics\_setpoint in 1% packet loss

Figure 40 shows the published data of the statistics topic: /statistics\_setpoint with 1% packet loss. This topic was enabled by the **offboard\_control** node. There were two parts: message age (the time starting when the message is delivered until it is received at the subscriber) and message period with the same measurement unit which is milliseconds (ms). The average, maximum, minimum, standard deviation, sample count was calculated in the moving windows (Please see Figure 41 to find the right data type). For the **message\_age** part, the average was 220ms which was lesser than 500ms (The setpoints must be streamed at greater than 2Hz (< 500ms) before entering the offboard mode and while the offboard mode is operational. Otherwise, the drone will exit the mode.). Therefore, the drone was still in operation. While the maximum was 2805ms, the minimum was 5ms. Here it took 463 as sample count with the standard deviation of 483.80ms. For the **message\_period** part, the average was 99.58ms which is close to the period of publishing setpoint (100ms). In this situation, this average was acceptable.

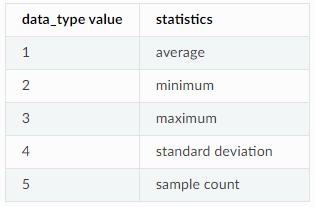


Figure 41: Statistics data\_type look-up table [23]

Figure 42 shows the setpoint message age average from 0% to 9% packet loss. The values of the setpoint message age average rose steadily from 196ms to 3541ms. With this very high message age average, the message loss happened certainly. At 2% packet loss, the average climbed to 780ms which was over 500ms. Therefore, starting at 2%, the drone was not operating. While the message age average increased gradually, the message period average (Figure 43) in the same packet loss remained stable around the set-up period (100ms). Although the message period average was good, the message age average exceeded the safe threshold that made the drone stopped flighting.

Figure 42: Setpoint message age average in reliable policy from 0 to 9% packet loss

Figure 43: Setpoint message period average in reliable policy from 0 to 9% packet loss

Whereas Figure 44 describes the setpoint message age average from 10% to 50% packet loss, Figure 45 describes the setpoint message period average in the same packet loss range. The data in both Figure 44 and 45 were a pair in the same source as Figure 40 has shown. The experiment was conducted from 0% to 90% packet loss, however, starting at 60%, the statistics data were given as Not a Number (NaN), which means the messages were not delivered. Therefore, these Figures show only from 10% to 50%. From 10% to 30% packet loss, whilst the message age average rose slightly from 3731ms to 8474ms, the message period average stabilized at around 100ms. After that, both numbers jumped dramatically: the message age soared to 57389ms, the message period shot up 457ms at 50% packet loss.

Figure 44: Setpoint message age average in reliable policy from 10 to 50% packet loss

Figure 45: Setpoint message period average in reliable policy from 10 to 50% packet loss

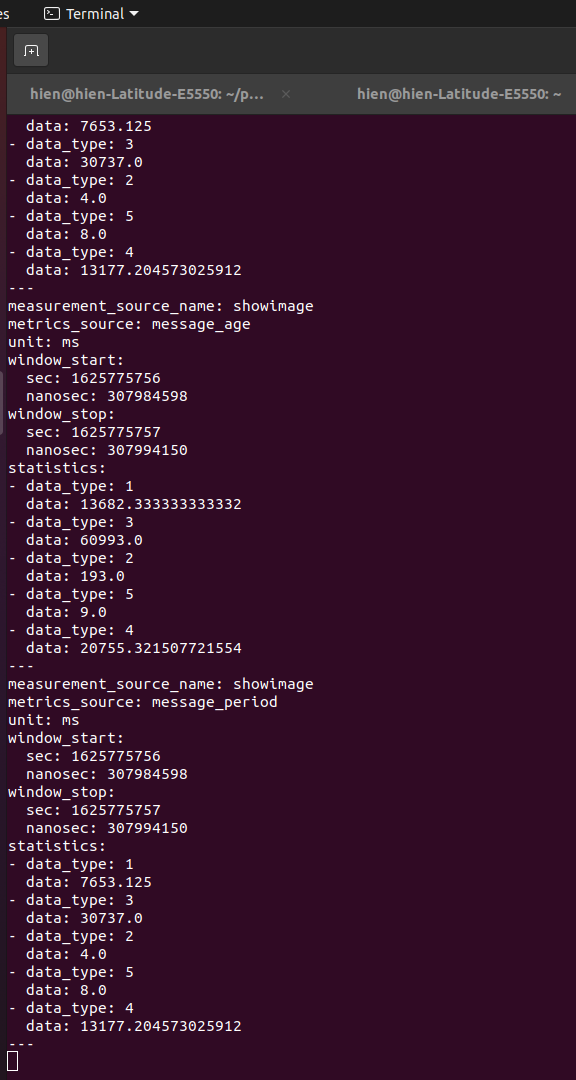


Figure 46: Reliable output terminal of topic /statistics\_image in 1% packet loss

Figure 46 demonstrates the output terminal of the topic /statistics\_image in 1% packet loss. In the **message\_period** part, although the set-up period was 33.33ms (30Hz), the message period average was 7653ms. That was a huge delay in just 1% packet loss in the default experiment. Even though the default size of images was just 240x320 pixels, the sample count was just 8 which implied that there were just a few images delivered. In the **message\_age** part, the message age average was even greater than their period average with 13682ms, which is not desirable. The output screen of the image stream in the GCS was just freezing for such a long time.

Figures 47 and 48 represent the image message age average and message period average in RELIABLE policy from 0% to 2% packet loss. Even though the packet loss range was from 0% to 90%, starting from 3%, the results were not given due to the inability to transfer the images. Both numbers in both figures escalated significantly during the conducted range. From 0% to 1% packet loss, while the message age average jumped from 2134ms to 13682ms, the message period average soared from 63.78ms to 7653ms. At 2% packet loss, the values rocketed to 117512ms for the message age and 23969ms for the message period. These high delays are not acceptable in our system.

The sensor data is published in a high frequency depending on the PX4 device, which has a very small size. Therefore, in the same condition, the message age and message period of the sensor data always had a better performance compared to setpoint and image. Figure 49 shows the output terminal of the topic /statistics\_sensor in 1% packet loss. The message age average and message period average are 23ms and 4ms respectively, which are great numbers.

The performance of sensor data is shown clearly in Figures 50 and 51 from 0% to 9% packet loss. Both indicators were very small and did not change much. While the sensor message age average fluctuated from 3ms to 23ms, the sensor message period average remained stable around 5ms. The statistics of sensor data from 10% to 40% (Figures 52 and 53) was still good, surprisingly, starting from 50%, the output data was Not a number, which means the messages were not delivered in this condition. Whilst the message age fluctuated from 8.7ms to 33ms, the message period stayed around 5.8ms.

Figure 47: Image message age average in reliable policy from 0 to 2% packet loss

Figure 48: Image message period average in reliable policy from 0 to 2% packet loss

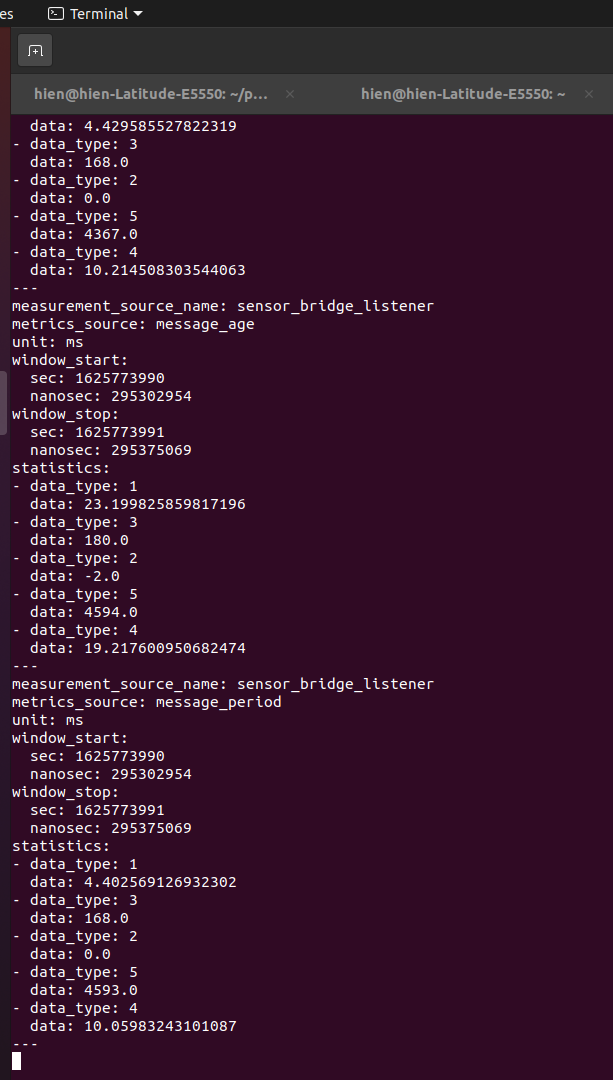


Figure 49: Reliable output terminal of topic /statistics\_sensor in 1% packet loss

Figure 50: Sensor message age average in reliable policy from 0 to 9% packet loss

Figure 51: Sensor message period average in reliable policy from 0 to 9% packet loss

Figure 52: Sensor message age average in reliable policy from 10 to 40% packet loss

Figure 53: Sensor message period average in reliable policy from 0 to 40% packet loss

To sum up, in this experiment, starting at 2% packet loss, the statistics data of setpoint and image was beyond the acceptance criteria. On the other hand, the sensor statistics are very good at 40% packet loss.

## 4.2. Experiment with the best-effort policy of QoS settings

In this section, we will observe the results of the **BEST\_EFFORT** of RELIABILITY policy. Like the first experiment, firstly the Setpoint statistics data are described, and then the Image statistics data, finally, the Sensor statistics data are discussed.

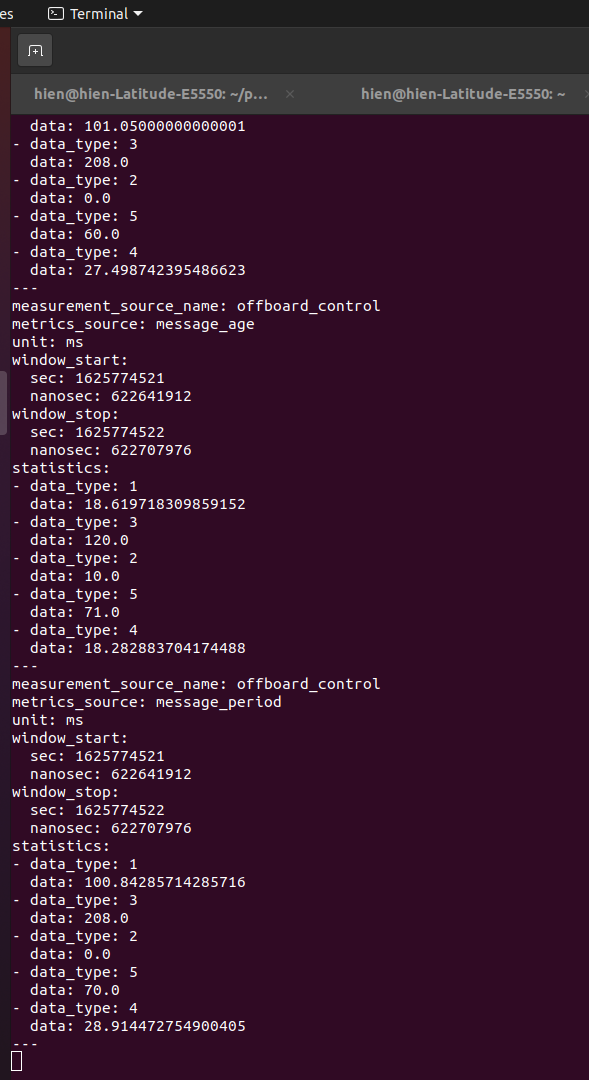


Figure 54: Best\_effort output terminal of topic /statistics\_setpoint in 1% packet loss

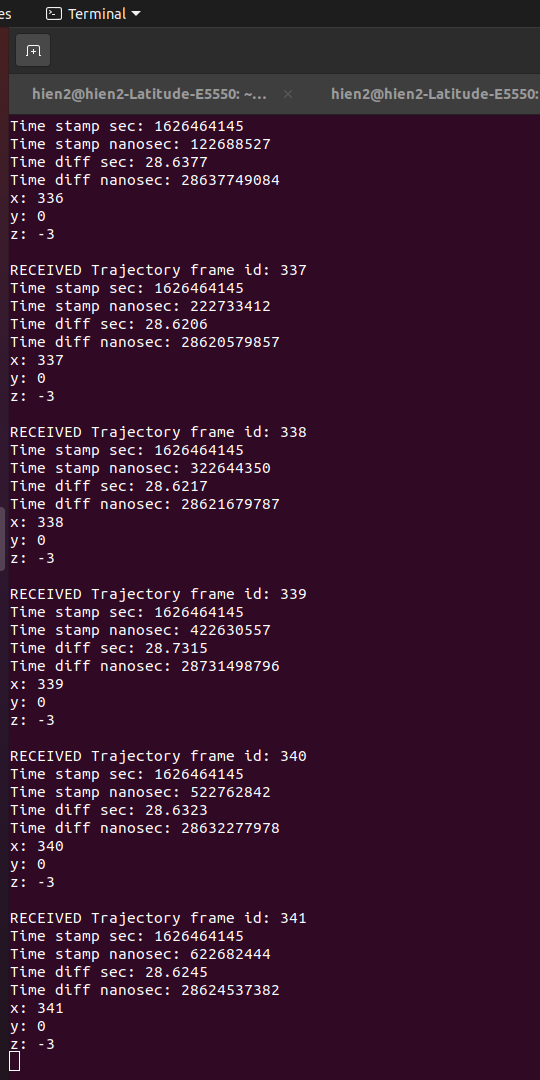


Figure 55: Best\_effort output terminal of offboard\_control node in 1% packet loss

Figure 56: Setpoint message age average in best\_effort policy from 0 to 9% packet loss

Figure 57: Setpoint message period average in best\_effort policy from 0 to 9% packet loss

As usual, the output of the topic /statistics\_setpoint in 1% packet loss with BEST\_EFFORT policy in Figure 54 is discussed. The setpoint message period was approximately 100ms which is equal to the set-up period. The maximum period number was just 208ms which just doubled the average. For the **message\_age** part, the setpoint message average was just 18ms which implied the message delay was too small. The interesting number is the standard deviation which was about 18ms which means the message age values were close to the average. The very small message age combined with the continuous frame\_id of Setpoints received by the **offboard\_control** node in Figure 55 confirms that there was no message loss with BEST\_EFFORT policy in 1% packet loss.

Figures 56 and 57 illustrate the setpoint message age average and period average from 0% to 9% packet loss. While the message age average noticed a downward trend, the message period average had an upward trend. Starting at 18ms at 0% packet loss, the message age average increased slightly to 19ms at 2% packet loss and then decreased gradually to 10.37ms at the end. On the other hand, the message period average went up steadily from 99.56ms at the beginning to 105.45ms at the end of the period. With these small messages age, we believe that there was no message loss in this case.

From 10% to 90% packet loss, the setpoint message age average and message period average noticed different trends (Figures 58 and 59). Whilst the message age average fluctuated from 4.4ms to 10.12ms, the message period average had an upward trend. Beginning at 105.92ms at 10% packet loss, the message period average increased slightly to 111.7ms at 40% packet loss, then it rose significantly to 200ms at 50% packet loss, before went up steadily to 249ms at the end of the period.

Figure 58: Setpoint message age average in best\_effort policy from 10 to 90% packet loss

Figure 59: Setpoint message period average in best\_effort policy from 10 to 90% packet loss

Coming to the image statistics data, firstly, the output terminal of topic /statistics\_image in 1% packet loss (Figure 60) is explained. The message period average was 4872ms and the sample count was just 20, which implies that there were several message losses in this condition. Surprisingly, the message age average was only 22.33ms that is very small compared to the message period.

Starting at 3% packet loss, the message results were Not a number that implicated the inability to transfer images. Figures 61 and 62 show the image message age average and period average from 0% to 2% packet loss. Both figures noticed upward trends. While the message age average increased slightly from 20ms to 22ms during the period, the message period average rocketed rapidly from 60ms at 0% packet loss to 15385ms at the end. With that high of delay, several images could not be delivered.

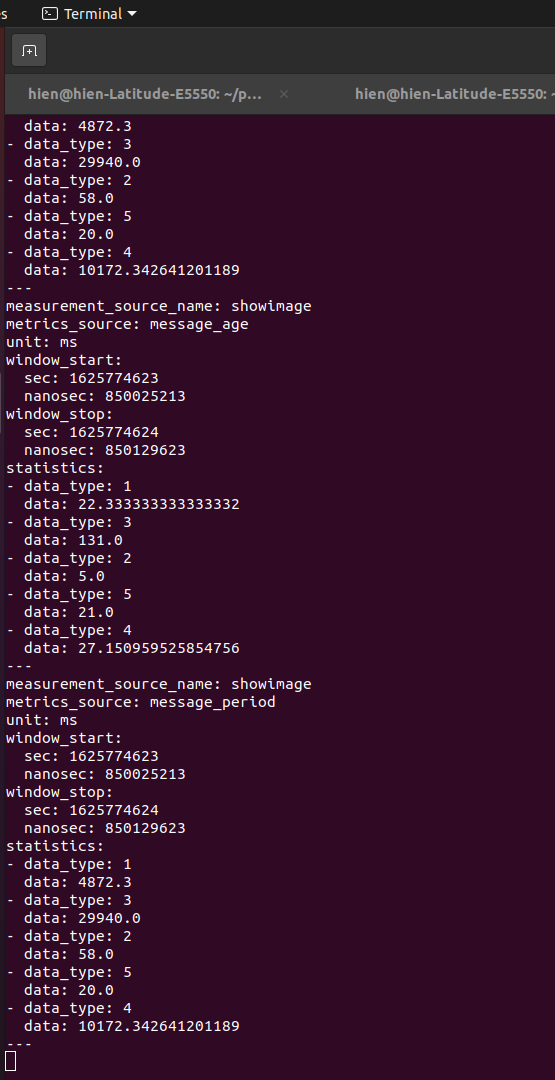


Figure 60: Best\_effort output terminal of topic /statistics\_image in 1% packet loss

Figure 61: Image message age average in best\_effort policy from 0 to 2% packet loss

Figure 62: Image message age average in best\_effort policy from 0 to 2% packet loss

Finally, the sensor statistics data are now explained. Figure 63 illustrates the output terminal of the topic /statistics\_sensor in 1% packet loss. Both the message age average and the message period average were very small with 7.35ms and 4.82ms respectively. Along with the high sample counts of around 6835, which implied that there was no message loss in this situation.

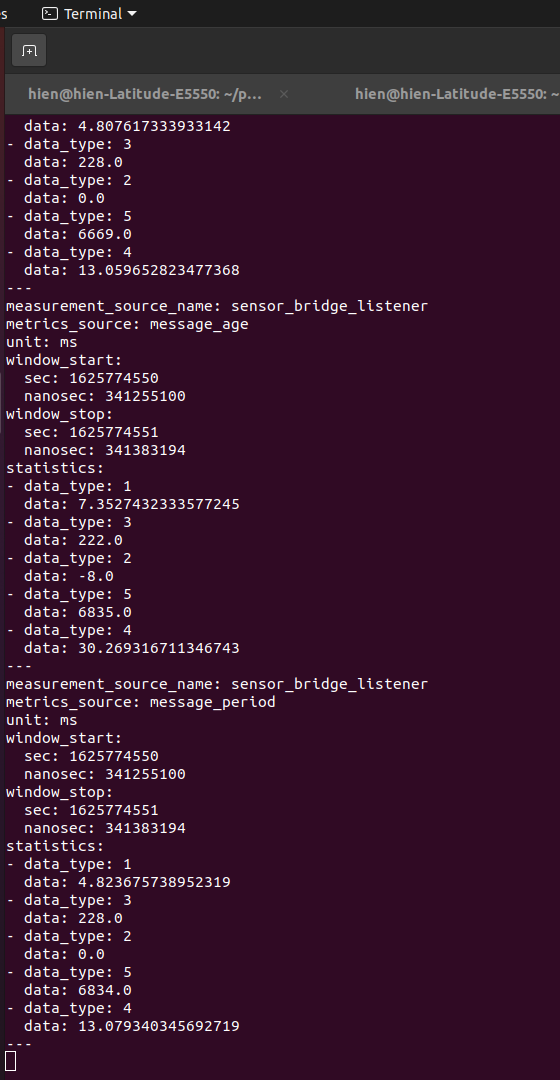


Figure 63: Best\_effort output terminal of topic /statistics\_sensor in 1% packet loss

Figure 64: Sensor message age average in best\_effort policy from 0 to 9% packet loss

Figure 65: Sensor message period average in best\_effort policy from 0 to 9% packet loss

From 0% to 9% packet loss, the sensor message age average and message period average noticed the same pattern which was the upward trend (Figures 64 and 65). Both figures fluctuated slightly from 0% to 5% packet loss. While the number of the message age average was between 7.3ms and 16.34ms, the number of the message period average was between 4.7ms and 5.2ms. From 5% to 9% packet loss, both indicators went up gradually; for the message age average, it increased from 13ms to 16.7ms; for the message period average, it grew from 4.9ms to 4.98ms. These small ages and periods prove the good performance in transferring messages in this period.

Figure 66: Sensor message age average in best\_effort policy from 10 to 90% packet loss

Figure 67: Sensor message period average in best\_effort policy from 10 to 90% packet loss

Figures 66 and 67 describe the sensor message age average and message period average from 10% to 90% packet loss. Both numbers rose gradually in the whole period. Whilst the message age average increased slowly from 17.6ms at 10% packet loss to 25.38ms at the end, the message period average went up steadily from 5.01ms at the beginning to 16.67ms at the end of the period. Even in the very high packet loss situation, the sensor statistics data was really good.

## 4.3. Comparison

In this section, the statistics data in the two experiments (the RELIABE and BEST\_EFFORT of RELIABILITY policy of QoS) in the same packet loss percentage are put into the same figures to make the comparison between them.

Initially, the setpoint results data are discussed. Figure 68 describes the setpoint message age average in both experiments from 0% to 9% packet loss. There was a clear distinction between RELIABLE and BEST\_EFFORT shown in the figure. While the BEST\_EFFORT numbers fluctuated from 11ms to 19ms, the RELIABLE numbers increased steadily from 196ms to 3541ms. The message age average of RELIABLE was from 10 times to 40 times more than those numbers of BEST\_EFFORT. In the same span of packet loss, the setpoint message period in both cases was similar, despite the numbers of the BEST\_EFFORT case were a little larger than those of the RELIABLE case (Figure 69). Whilst the message period average of the BEST\_EFFORT case was around 104ms, the message period average of the RELIABLE case was about 100ms. To sum up, we can see a much better performance in terms of the setpoint transmission performance of BEST\_EFFORT than RELIABLE. With the very small message age average and the desirable message period average, we can say that there was also no message loss in the BEST\_EFFORT experiment.

From 10% to 90% packet loss, the superior of BEST\_EFFORT in lossy wireless networks was shown even more crystally. Figure 70 gives information about the setpoint message age average in RELIABLE and BEST\_EFFORT policies from 10% to 90% packet loss. While the statistic of the BEST\_EFFORT case was just around 9ms during the whole span, that statistic of the RELIABLE case rocketed dramatically from 3731ms to 57389 which were 400 times to 6000 times more than that number of BEST\_EFFORT. Furthermore, starting at 60% packet loss, the data of RELIABLE were Not a Number. For the setpoint message period average, it had also the same pattern as his sibling (the message age average). In Figure 71, while the message period average of BEST\_EFFORT policy increased gradually from 105.92ms at the beginning to 249ms at the end (90% packet loss), that number of RELIABLE policy went up significantly from 99.87ms at 10% packet loss to 457ms at the end (50% packet loss). In total, from 10% to 90%, the message losses of the RELIABLE policy were too much. On the other hand, the message losses of the BEST\_EFFORT policy can be excluded from 10% to 40%, above this threshold, there might have a small number of message losses.

Figure 68: Setpoint message age average from 0 to 9% packet loss

Figure 69: Setpoint message period average from 0 to 9% packet loss

Figure 70: Setpoint message age average from 10 to 90% packet loss

Figure 71: Setpoint message period average from 10 to 90% packet loss

Opposing to the setpoint statistics data, the data of the image in both RELIABLE and BEST\_EFFORT policies were equivalent. Starting at 3% packet loss, the statistics data in both were given as Not a Number. In Figure 72, while the image message age average of the BEST\_EFFORT case was just around 22ms, that number of the RELIABLE policy jumped sharply from 2131ms to 117512ms. On the other hand, the message period average of both experiments had the same pattern, despite that the number of the RELIABLE was just a little larger than that of the BEST\_EFFORT (Figure 73). Starting at 63.78ms at 0% packet loss, the figure of the RELIABLE shot up steeply to 7653ms and then 23969ms. That number of the BEST\_EFFORT began at 60ms, increased suddenly to 4872ms before came to 15385ms. Overall, the performance of the BEST\_EFFORT policy was a bit better than that of the RELIABLE. However, starting at 3%, they were similar.

Figure 72: Image message age average from 0 to 2% packet loss

Figure 73: Image message period average from 0 to 2% packet loss

Finally, we are going to discuss the sensor statistics data now. Figure 74 represents the sensor message age average in the RELIABLE and BEST\_EFFORT policies from 0% to 9% packet loss. The range data of both cases were quite similar. While the data of the RELIABLE policy fluctuated from 3ms to 23ms in the whole span, the data of the BEST\_EFFORT policy fluctuated from 7.3ms to 16.34ms from the beginning to 5% packet loss, then it increased slightly from 13ms to 16.7ms. The sensor message period in the same range of packet loss in both cases was even more similar. In Figure 75, whilst the figure of the RELIABLE case was around 5ms, the figure of the BEST\_EFFORT was around 4.9ms. To sum up, in this situation, the performance of both is identical.

On the other hand, from 10% to 90% packet loss, there was a clear distinction between the experiments. In Figure 76, although the sensor message age average of the BEST\_EFFORT policy went up slightly from 17.6ms at the beginning to 25.38ms at the end (90% packet loss), the data of the RELIABLE policy fluctuated from 8.7ms to 33ms, ending at 40% packet loss. Starting at 40%, it was Not a Number. The sensor message period average statistics had the same template with their siblings. In Figure 77, while the figure of the BEST\_EFFORT policy rose gradually from 5.01ms at the beginning to 16.67ms at 90% packet loss, the data of the RELIABLE case was around 5.9ms from 10% to 40% packet loss. Overall, even though both statistics data in both experiments was similar from 0% to 9%, the data of the BEST\_EFFORT showed its superior to that of the RELIABLE from 10% to 90%.

Figure 74: Sensor message age average from 0 to 9% packet loss

Figure 75: Sensor message period average from 0 to 9% packet loss

Figure 76: Sensor message age average from 10 to 90% packet loss

Figure 77: Sensor message period average from 10 to 90% packet loss

# 5. Conclusion

## 5.1. Summary

In this thesis, we created a base experiment using PX4-Fast RTPS(DDS) bridge enabling the communication between PX4 and ROS2 for UAVs application. The experiment sends setpoints to the PX4 and also receives images and sensor data from PX4. We evaluated the RELIABLE and BEST\_EFFORT of the RELIABILITY policy of QoS settings in a lossy wireless network. The integration of PX4-Fast RTPS(DDS) bridge, Linux network traffic control (tc), and ROS 2 Topic Statistics is unique to this thesis and it was shown that the architecture of the experiment is effective to discover the characteristics of the RELIABLE and BEST\_EFFORT of the RELIABILITY policy of QoS settings. The message ages, message periods as well as message loss are observed and evaluated.

In conclusion, the setpoint and sensor statistics data of the BEST\_EFFORT policy showed a much better performance than those of the RELIABLE policy in a lossy wireless network. The message ages and message periods were small and below the maximum period of the PX4 devices (above the minimum frequency). The results also illustrated that the setpoints messages to the PX4 which is the most important are all transmitted without a single message loss. On the other hand, starting at 3% packet loss, the image transmission was not possible due to the large size of them in both policies. However, the statistics data of the BEST\_EFFORT case were a bit better than that of the RELIABLE policy.

## 5.2. Future work

This thesis’s work contributes towards evaluating the QoS settings in lossy wireless networks for Unmanned aerial vehicles. This helps UAVs to adapt better in situations that the network is not robust preventing failing the mission. Additional research would contribute to the performance of the UAVs.

### 5.2.1. Tuning of Additional QoS

There are still many other QoS policies needed to be evaluated, for instance, DURABILITY, DEADLINE, LIFESPAN, LIVELINESS, LEASE DURATION. Open Robotics says that even more QoS policies may be available in the future in ROS 2. Evaluating other QoS policies may help to create useful best practices for UAVs.

### 5.2.2. Use Case for a team of UAVs

In this thesis, only one UAV and one Ground Control Station (GCS) were used to evaluate the QoS policies. In reality, more UAVs are working as a team that communicates with each other and to the GCS. Making a new architecture for several UAVs and a GCS and then evaluate the QoS policies using it in lossy wireless networks make the results closer to the real applications.

### 5.2.3. Evaluation QoS policies in real UAVs

Simulation and real hardware are very different. After evaluating the QoS policies using the Gazebo simulation, testing on the real UAVs is also important. The UAVs have different processing power used in the Gazebo simulation. Furthermore, lossy wireless networks, in reality, are unpredictable compared to the simulated lossy networks. Therefore, evaluation QoS policies in real UAVs would give precise results.

# Bibliography

|  |  |
| --- | --- |
| [1] | B. Gerkey, "Why ROS 2?," Open Source Robotics Foundation, Inc., n.d. [Online]. Available: https://design.ros2.org/articles/why\_ros2.html. [Accessed 23 March 2021]. |
| [2] | "Robot Operating System," Wikipedia, n.a. [Online]. Available: https://en.wikipedia.org/wiki/Robot\_Operating\_System. [Accessed 9 April 2021]. |
| [3] | "ROS Introduction," Open Robotics, n.a. [Online]. Available: http://wiki.ros.org/ROS/Introduction. [Accessed 9 April 2021]. |
| [4] | "Nodes," Open Robotics, n.a. [Online]. Available: http://wiki.ros.org/Nodes. [Accessed 16 June 2021]. |
| [5] | "Topics," Open Robotics, n.a. [Online]. Available: http://wiki.ros.org/Topics. [Accessed 16 June 2021]. |
| [6] | "Understanding ROS 2 parameters," Open Robotics, n.a. [Online]. Available: https://docs.ros.org/en/foxy/Tutorials/Parameters/Understanding-ROS2-Parameters.html. [Accessed 17 June 2021]. |
| [7] | M. Hidalgo, "ROS Command Line Arguments," Open Source Robotics Foundation, Inc., [Online]. Available: https://design.ros2.org/articles/ros\_command\_line\_arguments.html. [Accessed 17 June 2021]. |
| [8] | "Use quality-of-service settings to handle lossy networks," Open Robotics, 2021. [Online]. Available: https://docs.ros.org/en/foxy/Tutorials/Quality-of-Service.html. [Accessed 17 June 2021]. |
| [9] | "Launching/monitoring multiple nodes with Launch," Open Robotics, 2021. [Online]. Available: https://docs.ros.org/en/foxy/Tutorials/Launch-system.html. [Accessed 17 June 2021]. |
| [10] | "Creating a launch file," Open Robotics, 2021. [Online]. Available: https://docs.ros.org/en/galactic/Tutorials/Launch-Files/Creating-Launch-Files.html. [Accessed 17 June 2021]. |
| [11] | I. Saito, "Packages," Open Robotics, 14 April 2019. [Online]. Available: http://wiki.ros.org/Packages. [Accessed 18 June 2021]. |
| [12] | "Creating your first ROS 2 package," Open Robotics, 2021. [Online]. Available: https://docs.ros.org/en/foxy/Tutorials/Creating-Your-First-ROS2-Package.html. [Accessed 18 June 2021]. |
| [13] | "RTPS/DDS Interface: PX4-Fast RTPS(DDS) Bridge," Dronecode, n.a. [Online]. Available: https://docs.px4.io/master/en/middleware/micrortps.html. [Accessed 19 June 2021]. |
| [14] | "Why Gazebo?," Open Source Robotics Foundation, [Online]. Available: http://gazebosim.org/. |
| [15] | "About Quality of Service settings," Open Robotics, 2021. [Online]. Available: https://docs.ros.org/en/foxy/Concepts/About-Quality-of-Service-Settings.html. [Accessed 22 June 2021]. |
| [16] | Y. Maruyama, S. Kato and T. Azumi, "Exploring the performance of ROS2," in *In Proceedings of the 13th International Conference*, Pittsburgh, Pennsylvania, 2016. |
| [17] | T. Kronauer, J. Pohlmann, M. Matthe, T. Smejkal and G. Fettweis, "Latency Analysis of ROS2 Multi-Node Systems," 11 June 2021. [Online]. Available: https://arxiv.org/abs/2101.02074. |
| [18] | P. Thulasiraman, Z. Chen, A. Bruce and B. Bingham, "Evaluation of the Robot Operating System 2 in Lossy Unmanned Networks," in *2020 IEEE International Systems Conference (SysCon)*, Montreal, QC, Canada, 2020. |
| [19] | "ROS 2 Foxy Fitzroy - Patch Release 4," [Online]. Available: https://github.com/ros2/ros2/releases/tag/release-foxy-20201211. |
| [20] | "px4\_msgs," [Online]. Available: https://github.com/PX4/px4\_msgs. |
| [21] | "PX4-ROS2 bridge," [Online]. Available: https://github.com/PX4/px4\_ros\_com. |
| [22] | H. Le, "operation\_control," [Online]. Available: https://github.com/LeQuangHien/operation\_control. |
| [23] | "ROS 2 Topic Statistics Tutorial (C++)," Open Robotics, 2021. [Online]. Available: https://docs.ros.org/en/foxy/Tutorials/Topics/Topic-Statistics-Tutorial.html. |