Fachhochschule Dortmund

University of Applied Sciences and Arts

(Master Embedded Systems for Mechatronics)

The Robot Operating System 2 (ROS 2) Evaluation

Research’s Thesis

Author: LE, Quang Hien

Matriculation Number: 7206974

Supervisor: Prof. Dr. Christof Röhrig

Co-Supervisor: Merlin Stampa

Date: 06/2021

# 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 HD images and sensor data from PX4. The throughputs, delays, message ages, message periods 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.

Table of Contents

[Abstract ii](#_Toc74819631)

[List of Abbreviations v](#_Toc74819632)

[List of Figures vi](#_Toc74819633)

[List of Tables vii](#_Toc74819634)

[1. Introduction 2 1](#_Toc74819635)

[1.1. Motivation 1](#_Toc74819636)

[1.2. Problem statement 2](#_Toc74819637)

[1.3. Structure 3](#_Toc74819638)

[2. Background 23 3](#_Toc74819639)

[2.1. ROS 2 basics 3](#_Toc74819640)

[2.1.1. Nodes 3](#_Toc74819641)

[2.1.2. Topics 4](#_Toc74819642)

[2.1.3. Parameters 6](#_Toc74819643)

[2.1.4. Workspace 6](#_Toc74819644)

[2.1.5. Packages 6](#_Toc74819645)

[2.1.6. Launch files 6](#_Toc74819646)

[2.2. RTPS/DDS Interface: PX4-Fast RTPS(DDS) Bridge 6](#_Toc74819647)

[2.2.1. Architectural overview 6](#_Toc74819648)

[2.2.2. Code generation 6](#_Toc74819649)

[2.2.3. uORB messages support 6](#_Toc74819650)

[2.2.4. Client (PX4/PX4-Autopilot) 6](#_Toc74819651)

[2.2.5. Agent in an Offboard Fast DDS interface (ROS-independent) 6](#_Toc74819652)

[2.3. Quality of Service 6](#_Toc74819653)

[2.3.1. Overview 6](#_Toc74819654)

[2.3.2. QoS policies 6](#_Toc74819655)

[2.3.3. Comparison to ROS 1 6](#_Toc74819656)

[2.3.4. QoS compatibilities 6](#_Toc74819657)

[2.3.5. QoS events 6](#_Toc74819658)

[2.3. Evaluation tools 6](#_Toc74819659)

[2.3.1. Linux network traffic control utility 6](#_Toc74819660)

[2.3.2. ROS 2 Topic Statistics 6](#_Toc74819661)

[2.4. Related works 6](#_Toc74819662)

[3. Experiments Implementation 15 6](#_Toc74819663)

[3.1. Architecture of the experiments 6](#_Toc74819664)

[3.2. Experiment with the default policy of QoS settings 6](#_Toc74819665)

[3.2.1. Introduction 6](#_Toc74819666)

[3.2.2. Implementation 6](#_Toc74819667)

[3.3. Experiment with the best-effort policy of QoS settings 7](#_Toc74819668)

[3.3.1. Introduction 7](#_Toc74819669)

[3.3.2. Implementation 7](#_Toc74819670)

[4. Results Discussion 10 7](#_Toc74819671)

[4.1. Experiment with the default policy of QoS settings 7](#_Toc74819672)

[4.2. Experiment with the best-effort policy of QoS settings 7](#_Toc74819673)

[4.3. Comparison 7](#_Toc74819674)

[5. Conclusion 5 7](#_Toc74819675)

[5.1. Thesis closure 7](#_Toc74819676)

[5.2. Outlook 7](#_Toc74819677)

[Bibliography 8](#_Toc74819678)

# 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](#_Toc76402025)

[Figure 2: Topic - Multiple Publisher and Multiple Subscriber [5] 5](#_Toc76402026)

[Figure 3: ROS 2 specific arguments in command line [7] 5](#_Toc76402027)

[Figure 4: ROS 2 configures argument using command line [8] 5](#_Toc76402028)

[Figure 5: Launch file example [10] 6](#_Toc76402029)

[Figure 6: How to run launch file [10] 7](#_Toc76402030)

[Figure 7: ROS 2 CMake packages with the minimum required contents [12] 7](#_Toc76402031)

[Figure 8: ROS 2 Python packages with the minimum required contents [12] 7](#_Toc76402032)

[Figure 9: Other directories in ROS 2 packages [11] 8](#_Toc76402033)

[Figure 10: A typical workspace in ROS 2 8](#_Toc76402034)

[Figure 11: microRTPS Bridge Architecture 9](#_Toc76402035)

[Figure 12: uorb\_rtps\_message\_ids.yaml file [13] 11](#_Toc76402036)

[Figure 13: Build PX4 SITL RTPS [13] 11](#_Toc76402037)

[Figure 14: Launching Client options [13] 12](#_Toc76402038)

[Figure 15: Starting the Client using UDP [13] 12](#_Toc76402039)

[Figure 16: The Agent command options [13] 12](#_Toc76402040)

[Figure 17: Starting the Agent using UDP [13] 12](#_Toc76402041)

[Figure 18: Installing iproute for tc 16](#_Toc76402042)

[Figure 19: Simulating 10% packet loss 16](#_Toc76402043)

# List of Tables

[Table 1: QoS policy setting options 20](#_Toc76402124)

[Table 2: QoS compatibilities of reliability and durability 21](#_Toc76402125)

[Table 3: QoS events 22](#_Toc76402126)

# 

# 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.

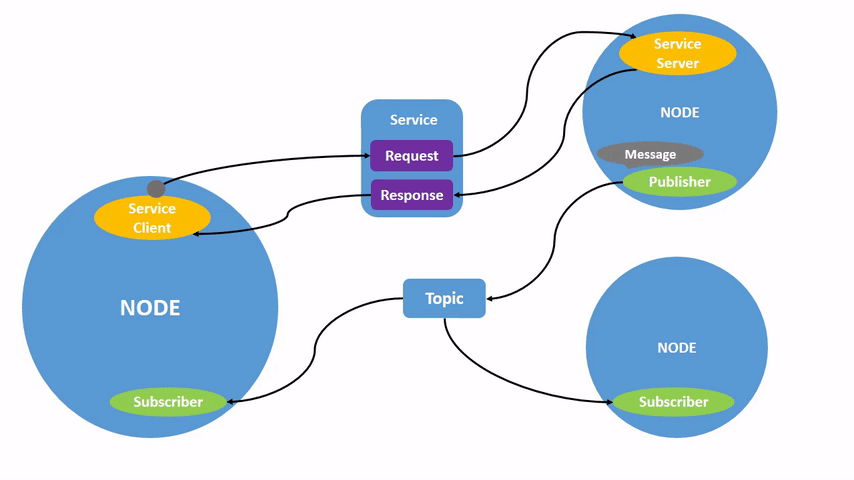


Figure 1: Nodes - Topic and Service [4]

### 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.

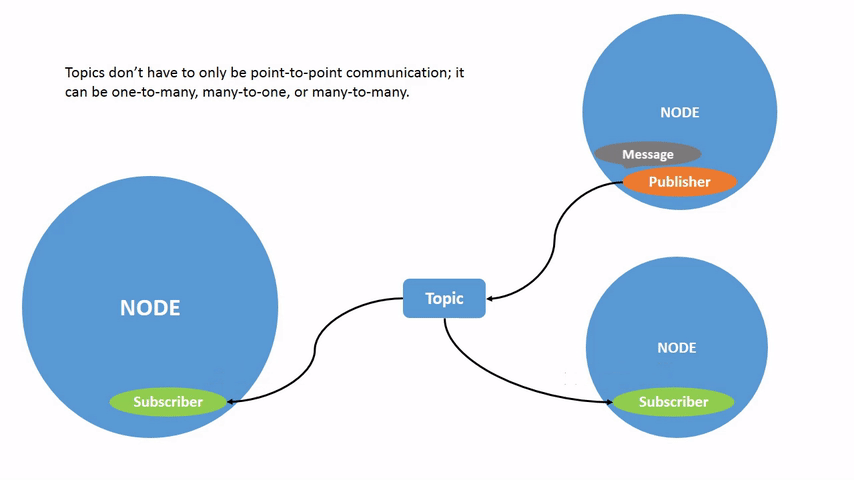


Figure 2: Topic - Multiple Publisher and Multiple Subscriber [5]

### 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.

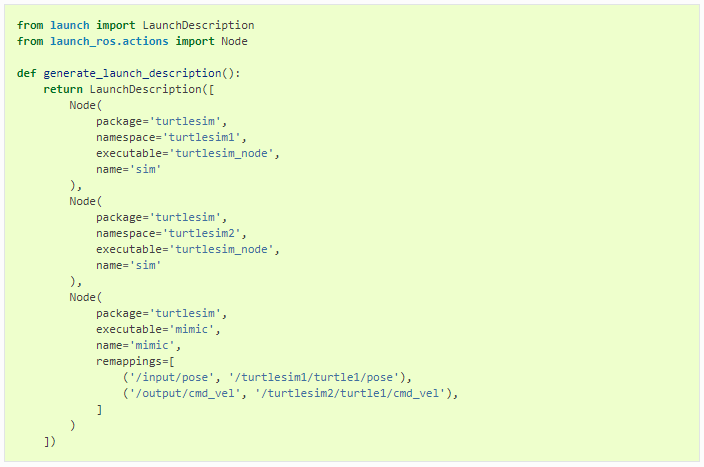


Figure 5: Launch file example [10]

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.

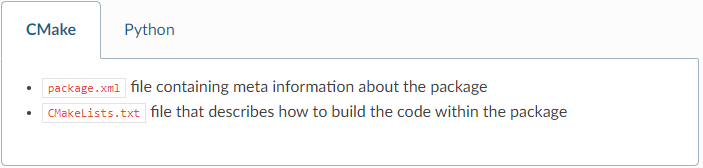


Figure 7: ROS 2 CMake packages with the minimum required contents [12]

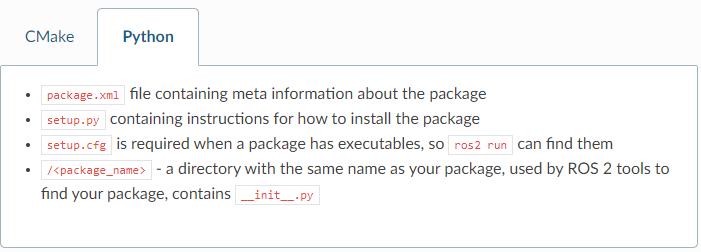


Figure 8: ROS 2 Python packages with the minimum required contents [12]

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.

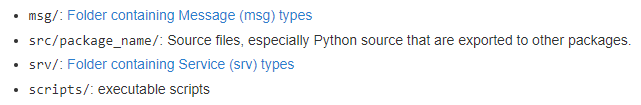


Figure 9: Other directories in ROS 2 packages [11]

### 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 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

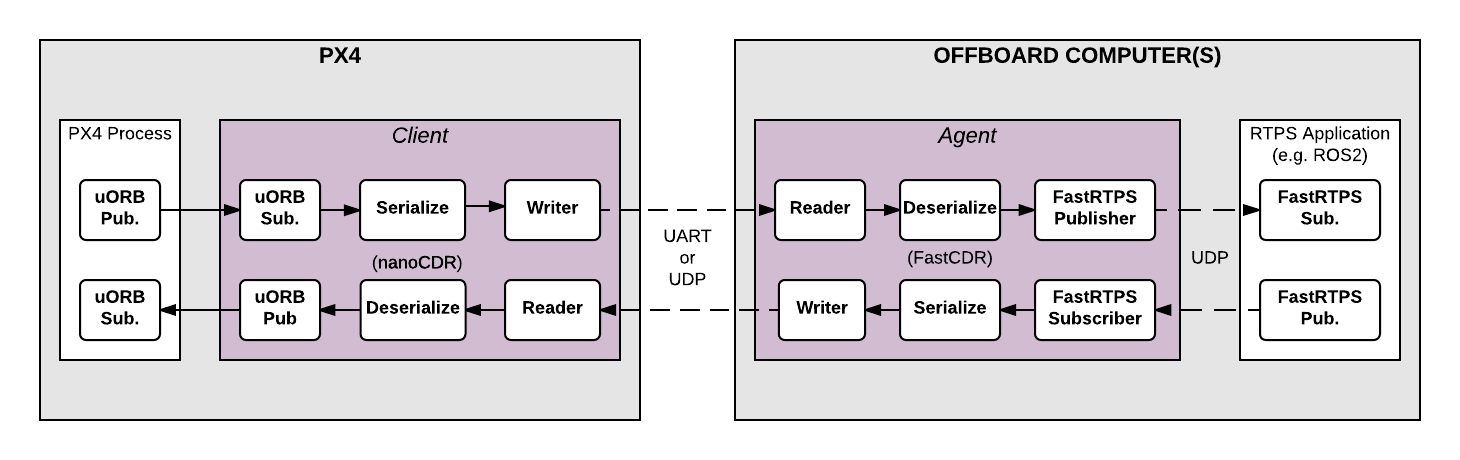


Figure 11: microRTPS Bridge Architecture

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 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. 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.

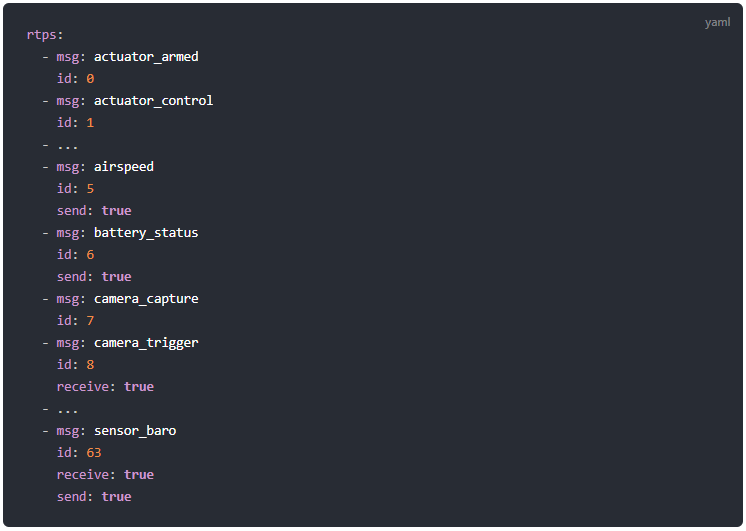


Figure 12: uorb\_rtps\_message\_ids.yaml file [13]

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

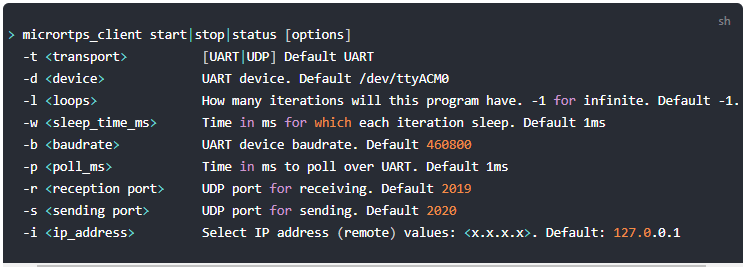


Figure 14: Launching Client options [13]

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.

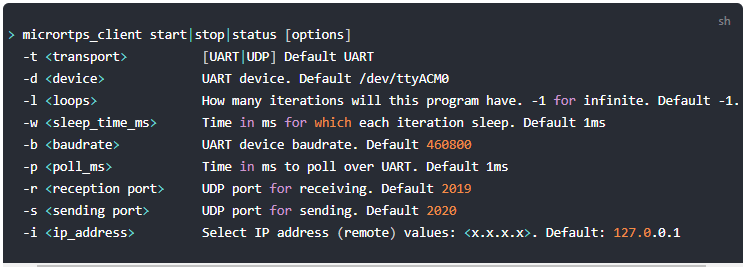


Figure 16: The Agent command options [13]

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 [14] |
| **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) [14]. |
| **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 [14]. |
|  | *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) [14]. |
| **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) [14]. |

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 [14]. 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 [14]. 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 [14]. |
|  | *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 [14]. |
|  | *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 in order 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 [15] 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 [16] 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 [17] 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. Experiments Implementation 15

## 3.1. Architecture of the experiments

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

### 3.2.1. Introduction

### 3.2.2. Implementation

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

### 3.3.1. Introduction

### 3.3.2. Implementation

# 4. Results Discussion 10

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

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

## 4.3. Comparison

# 5. Conclusion 5

## 5.1. Thesis closure

## 5.2. Outlook

# 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] | "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]. |
| [15] | Y. Maruyama, S. Kato and T. Azumi, "Exploring the performance of ROS2," in *In Proceedings of the 13th International Conference*, Pittsburgh, Pennsylvania, 2016. |
| [16] | 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. |
| [17] | 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. |