CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Cybernetics

Bachelor's Thesis

Using ROS 2 for High-Speed Maneuvering in Autonomous Driving

Martin Endler
Open Informatics – Artificial Intelligence and Computer Science

August 2022
https://github.com/pokusew/fel-bachelors-thesis
Supervisor: Ing. Michal Sojka, Ph.D.

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Endler Martin** Personal ID number: **483764**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Using ROS2 for High-Speed Maneuvering in Autonomous Driving

Bachelor's thesis title in Czech:

Použití ROS2 pro manévrování ve vysoké rychlosti v autonomním řízení

Guidelines:

The goal of this work is to improve real-time properties of the autonomous driving stack used by the CTU team for the F1/10 autonomous racing competition. The current stack is based on the Robot Operating System, whose first version (ROS1) is known for its problematic real-time properties.
1. Make yourself familiar with the ROS2 framework and study its differences to ROS1.
2. Port the CTU F1/10 autonomous driving stack from ROS1 to ROS2.
3. Evaluate the properties of the ported stack on the F1/10 platform (NVIDIA TX2).
Focus on real-time properties, temporal determinism, communication overheads etc.
4. Extend the autonomous driving stack for the ability to perform some high-speed maneuvers such as overtaking. The focus is on "high-speed", because such maneuvers are problematic without proper real-time support.
5. Propose and evaluate a method for off-line (or on-board) verification of the selected maneuvers (e.g. overtaking).

Bibliography / sources:

[1] ROS 2 Documentation (https://docs.ros.org/en/foxy/index.html)
[2] D. Casini, T. B. s, I. Lütkebohle, and B. B. Brandenburg, "Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling," in 31st Euromicro Conference on Real-Time Systems (ECRTS 2019), Dagstuhl, Germany, 2019, vol. 133, p. 6:1-6:23. doi: 10.4230/LIPIcs.ECRTS.2019.6.
[3] K. Osman, J. Ghommam, and M. Saad, "Guidance Based Lane-Changing Control in High-Speed Vehicle for the Overtaking Maneuver," J Intell Robot Syst, vol. 98, no. 3, pp. 643–665, Jun. 2020, doi: 10.1007/s10846-019-01070-6.

Name and workplace of bachelor's thesis supervisor:

Ing. Michal Sojka, Ph.D. Embedded Systems CIIRC

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **21.05.2021** Deadline for bachelor thesis submission: **15.08.2022**

Assignment valid until: **19.02.2023**

Ing. Michal Sojka, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgement / Declaration

First of all, I would like to thank my supervisor Ing. Michal Sojka, Ph.D. for his guidance, support, and immense patience.

Second, my thanks go to Ing. Jaroslav Klapálek for providing valuable information and advice regarding the F1/10 project.

Next, I would like to thank Bc. Tomáš Nagy for his help with all sorts of mechanical stuff, explaining AV algorithms, building a new CTU's F1/10 model, and for all his support during our joint internship at the University of Pennsylvania.

Last but not least, I would like to thank Prof. Rahul Mangharam for providing a stimulating environment during my internship at his lab at the University of Pennsylvania.

Finally, I would like to thank my family and close friends for always supporting me throughout my studies.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, August 15, 2022

.....

Abstract / Abstrakt

Performing high-speed maneuvers in autonomous driving is problematic without proper real-time support. At CTU, there is a team that competes in the F1/10 Autonomous Driving Competition with autonomous model cars. Their autonomous driving stack is based on ROS 1, which is not suitable for real-time applications.

The goal is to migrate this stack to ROS 2, which has been designed from the ground up to address many issues in this area. We propose tracing as an efficient way to analyze a running ROS 2 system and measure important properties.

We demonstrate the working of the migrated stack on the F1/10 model car and in the Stage simulator. We evaluate the communication latencies in the new stack using an extended version of ROS 2 tracing tools. Another result of our work is a publicly-available collection of setup guides, scripts, and documentation that covers various aspects of working with ROS. These guides have already helped several people.

We hope that the results of this thesis build a foundation that opens the way for the adoption of ROS 2 in the CTU's F1/10 stack, further improving its real-time properties, while making it more approachable by new students.

Keywords: ROS, ROS 1, ROS 2, ROS 2 migration, F1/10, Follow the Gap, autonomous model car, autonomous driving, NVIDIA Jetson TX2, LTTng, tracing, ros2_tracing

Manévrování ve vysoké rychlosti při autonomním řízení je problematické bez řádné podpory real-time na straně softwaru. Na ČVUT je tým, který se účastní soutěže F1/10 Autonomous Driving Competition s autonomními modely aut. Jejich software pro autonomního řízení je založen na ROS 1, který není vhodný pro real-time aplikace.

Cílem této práce je migrovat tento software na ROS 2, který byl od základu navržen s ohledem na real-time aplikace. Dále v práci navrhujeme tracing jako efektivní způsob analýzy běžícího systému na ROS 2 a měření důležitých parametrů.

Výsledky naší práci zahrnují software migrovaný do ROS 2, jehož funkčnost demonstrujeme na skutečném modelu F1/10 a v simulátoru Stage. Dále analyzujeme komunikační latence v migrovaném softwaru pomocí námi rozšířené verze tracing nástrojů pro ROS 2. Dalším výsledkem naší práce je veřejně dostupná sbírka návodů, skriptů a dokumentace, která pokrývá různé aspekty práce s ROS. Tyto návody již pomohly několika lidem.

Doufáme, že výsledky této práce vytvoří základ, který otevře cestu pro adopci ROS 2 v projektu F1/10 na ČVUT a zároveň jej zpřístupní novým studentům.

Klíčová slova: ROS, ROS 1, ROS 2, ROS 2 migration, F1/10, Follow the Gap, model autonomního auta, autonomní řízení, NVIDIA Jetson TX2, LTTng, tracing, ros2_tracing

Překlad titulu: Použití ROS 2 pro manévrování ve vysoké rychlosti v autonomním řízení

Contents /

Tables / Figures

Chapter 1
Introduction

Note: I severely underestimated the required time for writing this thesis text. During my work on this topic, I tried and made work tons of different things, leaving too little time for the final write-up. This is the reason why chapters 3 and 6 are incomplete. However, this thesis has a homepage[1] on GitHub, where an up-to-date revision of this text can be found.

Autonomous robots are successfully performing increasingly sophisticated tasks, often under challenging conditions. They must operate precisely and reliably because any hesitation or malfunction can cost human lives. At the same time, there is a strong desire among all robotics companies to shorten the time it takes to transform research prototypes into market-ready products. This is where the tools like the *Robot Operating System (ROS)* come in.

Since ROS was started in 2007 [1], it has gained great popularity and has become the standard in the robotics community. However, it has turned out that the original architecture (ROS 1) has some limitations concerning *performance, efficiency and real-time safety*, making it unsuitable for production deployments. Thus, a new version of ROS – **ROS 2** – has been designed from the ground up to allow more use-cases and solve many pain points of ROS 1 [2–3]. Missing features and incompatible packages have been slowing down the adoption of ROS 2 since its first public release in 2017. But in recent years, the situation has improved a lot, and the adoption of ROS 2 has accelerated [4]. Thus, now might be the right time to start migrating applications from ROS 1 to ROS 2 and benefit from the new possibilities.

Figure 1.1. One of the CTU's F1/10 Autonomous Model Cars.

[1] `https://github.com/pokusew/fel-bachelors-thesis`

At CTU, ROS 1 has been at the heart of the autonomous driving stack used by the CTU team for the *F1/10 Autonomous Racing Competition* for many years. High-speed maneuvers and racing in general are areas where many usually neglected details such as communication latency, jitter, and temporal determinism, suddenly play a significant role.

The goal of this thesis is to migrate a selected part of the CTU's F1/10 project from ROS 1 to ROS 2, taking advantage of its new features. The result should be a working port running on ROS 2 in a simulator and on a physical model car with an NVIDIA Jetson computing module. Also, an approach for analyzing the real-time properties of the running AV stack should be presented.

This thesis has two main parts. While the first three chapters cover the necessary background information and theory, the last three chapters present the contributions.

First, in Chapter 2, both versions of ROS are briefly described while highlighting the main differences between ROS 1 and ROS 2.

Second, Chapter 3 explores tools and approaches that can be used to measure execution times, latencies, jitter, communication delays, and other parameters throughout the ROS system. Tracing using LTTng is presented as an efficient way of analyzing all of the important runtime parameters.

The third and last missing piece of the background information, namely the up-to-date description of the CTU's F1/10 platform, is then presented in Chapter 4.

Building on the introduced concepts, Chapter 5 recounts the actual process of the migration, its results, and some of the encountered challenges along the way.

Chapter 6 focuses on the analysis of the ported stack. First, the `ros2_tracing` framework is extended with message flow analysis. Then, this extended tool is used to perform an analysis of communication latencies between different nodes.

Finally, **the achieved results** are summarized in the last Chapter 7.

1.1 Motivation / Why to migrate to ROS 2?

Before we move on to the next chapter, we would like to pause for a moment and explore some of the reasons for migrating to ROS 2.

As already stated in the Introduction and as we will elaborate in section 2.4, ROS 2 has been designed from the ground up, addressing some of the pain points of ROS 1, and paving the way for real-time applications. While this on its own might be a valid reason to pursue a migration to ROS 2, there are a few more reasons that can make the case for migration even stronger:

- ROS 1 will be deprecated soon (2025)
- There is an increasing number of ROS-2-only packages.
- While the latest ROS 1 release, Noetic Ninjemys, offers a decent set of features, the CTU's F1/10 platform is, in fact, based on a now-prehistoric and deprecated (but wildly popular at the time) ROS 1 Kinetic Kame. Instead of spending time making the codebase compatible with the latest ROS 1, it is better to migrate to ROS 2 straight away and benefit from its new features.

It is also fair to admit that there might exist valid reasons why not to migrate – they include increased complexity, greater out-of-the-box overhead, or missing (not ported) packages. Fortunately, ROS developers have worked hard to make those irrelevant.

Chapter 2
Robot Operating System

In this chapter, we describe both versions of ROS, ROS 1 and ROS 2, focusing on the latter in more detail. We also highlight the main differences.

The Robot Operating System (or ROS) is "an open-source, meta-operating system" [5] that consists of "tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot applications across a wide variety of robotic platforms" [6].

A typical ROS application is composed of loosely coupled processes – **nodes** – (potentially distributed across machines) that communicate with each other and work together to accomplish a certain goal (for example, autonomous vehicle control). Each node performs only a limited set of specific functions *(e.g., one node can read data from LiDAR, another node can implement an obstacle detection algorithm from the LiDAR data, while another node can use the detected obstacles to plan the trajectory, etc.).*

Such architecture greatly supports the separations of concerns and allows code reuse. That further enables efficient code sharing of common functionality among different projects with various applications. That, in fact, is one of the most significant features of ROS, as there are thousands of ROS packages provided by the ROS community [7]. Developers can focus on their application-specific problems while reusing code for common parts.

2.1 ROS Computation Graph

At runtime, ROS nodes and their communication interactions form the so-called "ROS Computation Graph" [8]. The nodes are represented by the graph's vertices, while the edges depict the communication interactions.

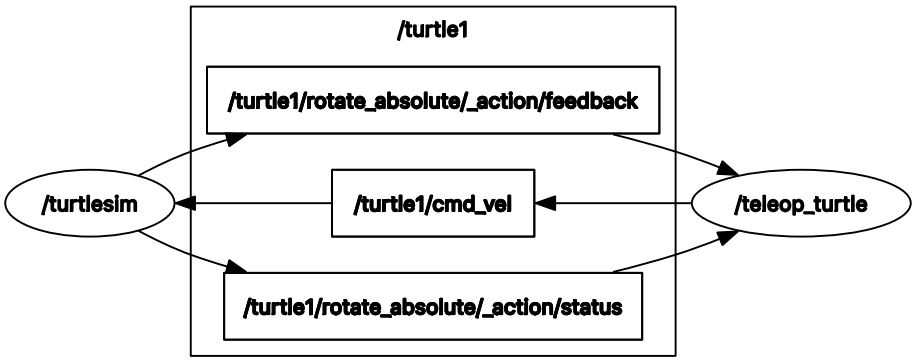

Figure 2.1. A simple ROS Computation Graph. Two nodes (represented as ellipses) (`/turtlesim` and `/teleop_turtle`) are running. The rectangles represent topics. The application is a part of the `turtlesim`[1] package.

[1] `https://index.ros.org/p/turtlesim/`

2.1.1 ROS Communication Primitives

ROS provides several *communication primitives* that can be used by nodes:

1. **messages and topics** – publish/subscribe
   Nodes (publishers) can publish messages to a named topic. Other nodes (subscribers) can subscribe to a topic and receive the published messages.
2. **services** – synchronous RPC (Remote Procedure Call) (server/client)
   Nodes (service servers) can provide services. Services have names. Other nodes (service clients) can invoke/call services and synchronously get results.
3. **actions** – asynchronous preemptible RPC with continuous feedback (server/client)
   Node (action servers) can provide actions. An action is a preemptible task that has a goal, can provide continuous feedback, and returns a result if it is not canceled. Other nodes (action clients) can invoke an action (request a goal and subscribe for its feedback and result).

The communication is strongly typed, and ROS provides an interface description language (IDL) for describing message types (used for pub/sub), service types, and action types.

2.2 ROS Common Concepts

In addition to the Computation Graph and communication possibilities, both versions of ROS share many additional common concepts, some of which are described below:

- **Package** – Package – a container for code, IDL files (messages, services, actions), configuration files, or anything else. It is "the most atomic build and release item" that groups together common functionality that can be easily shared and reused [8]. Each package has a package manifest file `package.xml` that provides "metadata about a package, including its name, version, description, license information, dependencies, etc.". There are currently 3 package manifest formats, which are defined in REP 127[2], REP 140[3], and REP 149[4], respectively.
- **Distribution** – "a versioned set of ROS packages [9]. ROS 1 distributions are listed at [9], ROS 2 distributions are listed at [10].
- **Workspace** – a directory containing packages that are built together using a ROS build tool (such as `catkin_make`, `catkin_make`, `catkin_make_isolated`, `catkin_tools`, `colcon`) and a build system (such as `catkin` or `ament`).
- **Graph Resource Names** – "a hierarchical naming structure that is used for all resources in a ROS Computation Graph, such as Nodes, Parameters, Topics, and Services" [8].
- **Package Resource Names** – a simplified method of "referring to files and data types on disk" [8]. It consists of the name of the package that the resource is in plus the name of the resource. For example, the name `std_msgs/String` refers to the `String` message type in the `std_msgs` package.
- **ROS client library** – a collection of code that simplifies the task of implementing a ROS node in a certain programming language. "It takes many of the ROS concepts and makes them accessible via code" [11]. It provides an API (functions, methods, classes) that allows a user program to interact with other nodes (using pub/sub,

[2] `https://ros.org/reps/rep-0127.html`
[3] `https://ros.org/reps/rep-0140.html`
[4] `https://ros.org/reps/rep-0149.html`

services, actions, and parameters) and do other common things. The main ROS client libraries are for C++ and for Python. The internal architecture of client libraries differs greatly between ROS 1 and ROS 2.

2.3 ROS 1

ROS 1 is the original version of ROS that dates back to 2007 [1]. Version 1.0 was published in 2010. Since then, 13 ROS 1 distributions have been released [9]. The latest ROS 1 version, Noetic, was released in 2020. There are no plans to release any new ROS 1 versions besides Noetic. Its support will end in May 2025. After that date, ROS 1 will be effectively deprecated.

Full documentation of ROS 1 can be found at [12].

2.3.1 Architecture

In ROS 1's Computation Graph, there must always be a special node called *ROS Master*. It provides name registration and lookup services to the rest of the Computation Graph. It coordinates the communication among the nodes. That includes graph changes notifications and establishing connections between nodes. It offers two XML-RPC-based APIs – Master API and Parameter Server API.

ROS 1 has a concept of central key/value data storage called *Parameter Server*. It is a part of ROS Master (Parameter Server API). a key/value pair is called a parameter. All nodes in the Computation Graph can get and manipulate the key/value data stored in the Parameter Server. The parameters can be used as configuration storage for nodes, which allows easy altering of the system (nodes') behavior at runtime.

ROS 1 offers two underlying data transport protocols – *TCPROS* and *UDPROS*. As the names suggest, they are based on TCP and UDP, respectively.

All concepts are implemented directly (natively) in ROS 1 Client Libraries. The two main client libraries are **roscpp** (for C++) and **rospy** (for Python). Their performance and features' availability varies greatly. While the C++ ROS 1 Client Library implements all ROS 1 features and provides high performance, the Python ROS 1 Client Library lacks some features and provides worse performance. Even when a feature is available in both libraries, the actual implementation often comes with minor differences that might not be expected.

2.3.2 Build System

ROS 1 uses catkin build system[5]. Catkin supports CMake packages that use special catkin CMake macros. The actual build is controlled by a build tool. ROS 1 catkin workspace can be built using different build tools – `catkin_make`, `catkin_make_isolated`, `catkin_tools`.

2.3.3 CLI

The CLI is composed of several commands that cover all ROS 1 features. These include, for example `rostopic`, `rosservice`, `rosparam`, `rosmsg`, `rosrun`, `roslaunch`, etc. [13].

[5] In older ROS 1 distributions, rosbuild was used. It is also (theoretically) possible to use ROS 2 build tool colcon to build ROS 1 workspace.

2.3.4 Launch System

While nodes can be started manually (via running the corresponding executables), it may be cumbersome in a complex system. For this reason, ROS 1 allows describing the system using a special XML file. Then the command `roslaunch` handles the process of starting up all nodes and supplying correct arguments to them [14].

2.4 ROS 2

"Since ROS was started in 2007, a lot has changed in the robotics and ROS community. The goal of the ROS 2 project is to adapt to these changes, leveraging what is great about ROS 1 and improving what is not." [6]

Full documentation of the latest ROS 2 release can be found at [6].

2.4.1 Architecture

The architecture of ROS 2 was designed from the ground up, addressing issues of ROS 1 [2]. The newly designed architecture should address new use-cases as well as many issues from ROS 1:

- Truly distributed system (no master node).
- Support for real-time.
- More nodes in one process (Composable nodes).
- Better support for communication in non-ideal networks.
- Small embedded platforms support.

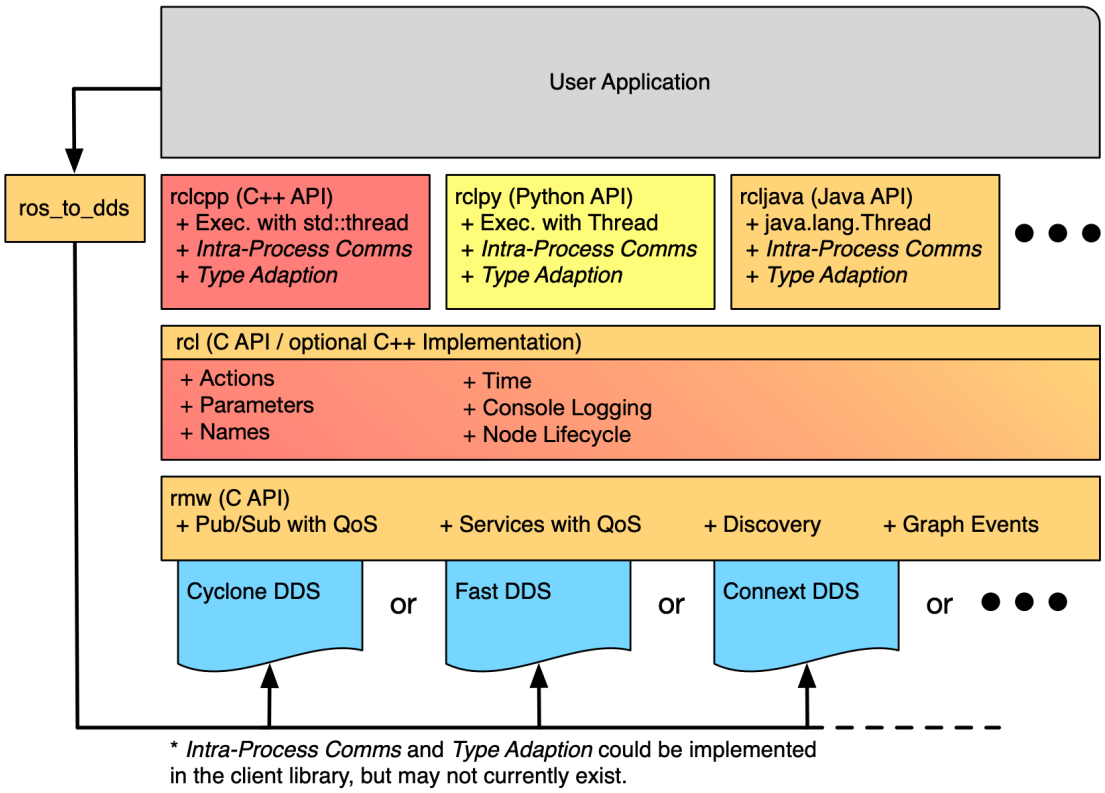

Figure 2.2. ROS 2 Architecture [15].

While ROS 1's communications stack is built almost entirely from scratch, ROS 2 relies on *Data Distribution Service* (DDS). DDS is "a middleware protocol and API standard for data-centric connectivity from the Object Management Group (OMG). It provides low-latency data connectivity, extreme reliability, and a scalable architecture for Internet of Things applications need" [16].

ROS 2 Client Libraries have a different architecture compared to the ROS 1 ones. Instead of reimplementing all the features in all the programming languages separately, the common functionality is implemented in the *rcl* library that exposes a C API. Client Libraries then use the *rcl* library and implement the rest of the features on top of it (in particular, language-dependent features, such as threading and execution model).

Because ROS 2 supports different DDS implementations from different vendors [17], the rcl library cannot directly communicate with the DDS implementation. Instead, there is an abstraction layer called *rmw* (ROS middleware) that provides unified access to DDS. The specific DDS implementation can even be changed dynamically at runtime.

The whole relationship among different parts of ROS 2 Client Libraries is shown in Figure 2.2. Additional detailed information can be found at [15].

2.4.2 Build System

ROS 2 uses ament as a build system and colcon as a build tool. Ament supports three types of packages: CMake with `ament_cmake`, pure Python packages (Python setuptools based), and pure CMake packages. Colcon always builds all packages in isolation and in the correct order. For ensuring the correct build order, a dependencies graph is constructed, which must be a directed acyclic graph in order for the workspace to be buildable. It also tries to parallelize the build up to the level that mutual dependencies among packages allow.

2.4.3 CLI

It is very similar to ROS 1 CLI, but instead of having multiple commands, ROS 2 has one central command, `ros2,` with several subcommands.

2.4.4 Launch System

The Launch System has been completely redesigned to support the new concepts in ROS 2. It is implemented in Python, and it provides a way to *declaratively* describe the system to launch [18–19]. Launch files can be written in Python, XML, or YAML. Python-based launch files represent the most powerful and comfortable[6] method of describing a system to launch. In fact, the other supported formats, XML and YAML, directly map to the Launch System's Python API.

Because of their declarative nature, Python-based launch files might be quite verbose and even might seem counter-intuitive and restrictive at first. But the declarative approach brings many advantages. "By separating the declaration of an action from the execution of an action, tools may use the launch descriptions to do things like visualize what a launch description will do without actually doing it." [19] This is used for example by `ros2 launch` command to determine and print launch arguments of a launch file. It is also the reason why it is possible to map some of the Launch System declarative APIs to declarative languages such as XML and YAML.

[6] Since Python-based launch files are just Python code, one can take advantage of IDEs' powerful auto-completion and coding assistance features.

2.5 Summary of Differences between ROS 1 and ROS 2

ROS 2 has better architecture and offers more features. Thanks to DDS and its fine-grained QoS, ROS 2 handles communication in non-ideal networks better than ROS 1. Furthermore, the ROS 2 client libraries offer better control over code execution and threading as they support writing custom executors. ROS 2 architecture is designed with real-time support in mind.[7]

The following table summarizes the most notable differences between ROS 1 and ROS 2:

Category	ROS 1	ROS 2
Supported Platforms	Only Ubuntu officially supported.	Ubuntu, Windows, and macOS officially supported. Yet, a lot of packages work only on Ubuntu.
Client Libraries	Written independently in each language.	Sharing a common underlying C library (rcl).
Transport	TCPROS or UDPROS	Handled by DDS which offers fine-grained QoS.
Real-Time Support	No. Not part of the design.	Yes. One of the design goals.
Runtime Node Composition	No. But Nodelets can be used.	Yes. Composable Nodes.
Threading and Execution Model	not much customizable	granular execution models, custom executors
Parameters	global parameter server	parameters per node (no global parameter server), out-of-the-box dynamic_reconfigure-like features
Communication Primitives	pub/sub, services, actions (not natively, but via `actionlib`)	pub/sub, services, actions
IDL	`.msg/.srv`	`.msg/.srv/.action` + extended features such as constraints
Launch System	XML-based launch files	extensible, Python-based, XML and YAML supported as well
Build	`catkin` + `catkin_make` or `catkin_make_isolated`	`ament` + `colcon`

Table 2.1. Summary of ROS 2 features compared with ROS 1. Inspired by [3].

[7] Although the actual real-time properties may differ dramatically based on various configuration and threading/execution model choices.

Chapter 3
Methods for Evaluating ROS Applications

The current F1/10 algorithms sometimes do not work as well as expected. That may be caused by variation in latencies (jitter) throughout the software pipeline (and non-deterministic runtime in general).

There have been many studies that focused on analyzing ROS 2 properties in this area. However, most of them have been using synthetic or simplified systems, because it is hard to collect the required data without affecting the running system and without the need for extensive changes in the evaluated codebase.

This problem prompted the creation of `ros2_tracing` [20] framework, which brings an efficient way to collect and analyze runtime metrics of real ROS 2 systems. It relies on LTTng (Linux Trace Toolkit Next Generation) for efficiently collecting runtime events. It adds instrumentation to the core ROS 2 libraries (DDS, rmw, rcl, rclcpp). Because the framework has become a part of ROS 2 core, instrumentation points are now shipped in all of the core ROS 2 libraries (although they are disabled unless `tracetools` package is rebuilt on a system where LTTng is present).

Nevertheless, the hard part is getting useful results from the collected tracing events. With some effort, it is possible reconstruct messages flow across the system, including casual relationships, opening a way for interesting analyzes [21].

We will use the `ros2_tracing` in Chapter 6 to evaluate some properties of our migrated ROS 2 stack.

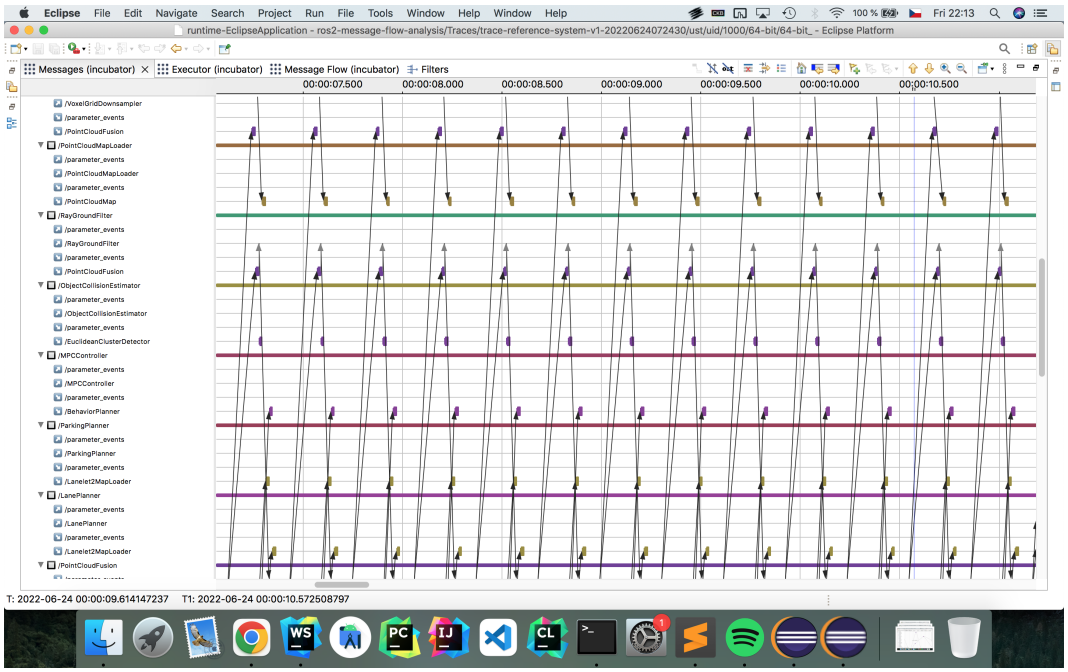

Figure 3.1. A message flow reconstruction as implemented in [21]. We were able to recreate the results following the steps the paper.

Chapter 4
CTU's F1/10 Platform

The F1/10 platform is a scaled-down (1:10) model of an autonomous car that originates from the F1/10 Autonomous Racing Competition[1]. Thanks to its affordability and similarity to a real car, the platform can be easily used for developing, testing, and verifying autonomous driving systems and related algorithms.

At CTU, multiple F1/10 models have been created and used [22–24]. In this thesis, we focus on the latest iteration of the design represented by the models codenamed `tx2-auto-usa`[2] and `tx2-auto-3`, which are depicted in Figure 4.1 and Figure 4.2, respectively. This chapter aims to provide an up-to-date functional description of their hardware and software components. Such a description is a prerequisite for a successful migration to ROS 2.

Figure 4.1. The CTU's `tx2-auto-usa` F1/10 Model Car based on NVIDIA Jetson TX2.

[1] `https://f1tenth.org/`

[2] This new model was designed by a fellow student, Tomáš Nagy. The main difference compared to the `tx2-auto-3` is its chassis design (components' placement and mounting). The `usa` suffix in the model name refers to the fact that this model was assembled in the USA during our (my and Tomáš's) internship at the University of Pennsylvania.

Figure 4.2. The CTU's `tx2-auto-3` F1/10 Model Car based on NVIDIA Jetson TX2.

4.1 Hardware Stack

The platform is based on an off-the-shelf RC car model from Traxxas (i.e., Traxxas Slash, but different ones can be used as well) with added components that enable autonomous operation (sensors, computers). The list below provides a quick overview of those components, which are then described in detail in the following subsections. The Figure 4.3 shows the functional relationships among them.

- **NVIDIA Jetson TX2** – The main computing unit that runs the autonomous driving stack.
- **VESC (Enertion FOCBOX VESC-X)**– Electric Speed Controller (ESC) that controls the BLDC motor.
- **Teensy 3.2** – a microcontroller (MCU) for controlling the steering and handling the communication with an RC Transmitter (Manual Control). It also implements an independent emergency stop (eStop).
- **Hokuyo UST-10LX** – LiDAR
- **SparkFun 9DoF Razor IMU M0** – IMU

4.1.1 Chassis and Powerboard

While the main chassis, BLDC drive motor, steering servo, and RC receiver from the Traxxas model are preserved, the ESC is replaced by a VESC.

Further, a structure for mounting the sensors and the main computing unit is added. While `tx2-auto-3` uses a two-layer design – one laser-cut plate which is mounted on the original chassis using standoffs; `tx2-auto-usa` features a much more sophisticated and streamlined design consisting of multiple 3D-printed parts. The `tx2-auto-usa`'s design allows much easier battery replacement.

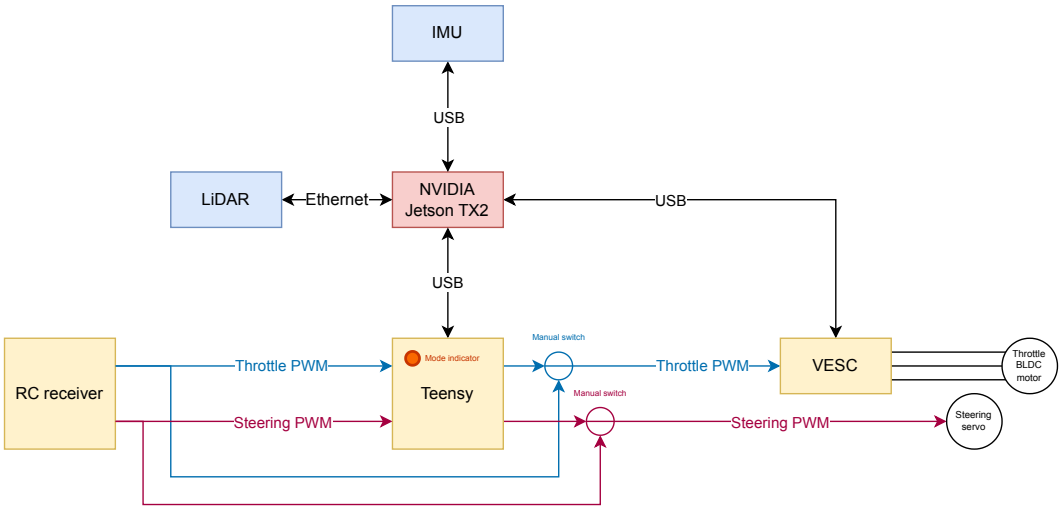

Figure 4.3. Overview of main components with depicted methods of communication. The yellow blocks are part of the low-level system, which always works independently of the main high-level system (NVIDIA Jetson TX2).

Finally, a powerboard is needed to power the AV components. For that purpose, a custom powerboard has been designed. One of the latest iterations was created as a part of [23].

4.1.2 NVIDIA Jetson TX2

The NVIDIA Jetson TX2 forms the `brain` of the car. It processes data from all sensors and decides what to do. In other words, it runs the autonomous driving pipeline, which usually includes perception, localization (state estimation), planning, and control. Such a task requires a powerful device. Additional requirements include power-efficient operation (because of the limited capacity of batteries) and small form factor (because of limited space in the chassis). Fortunately, NVIDIA Jetson TX2 fulfills all of these requirements.

NVIDIA Jetson TX2 is a very powerful but power-efficient embedded system (SoC) designed for autonomous machines. It is distributed as a very compact module with a standardized board-to-board connector. This way, it can be used in applications that might require different physical IO interfaces. On our car, we use the Orbitty Carrier board from Connect Tech, which provides a suitable form factor for our use case. The Table 4.1 lists some of Jetson TX2's key specs.

GPU	256-core NVIDIA Pascal™ GPU architecture with 256 NVIDIA CUDA cores
CPU	Dual-Core NVIDIA Denver 2 64-Bit CPU Quad-Core ARM® Cortex®-A57 MPCore
Memory	8 GB 128-bit LPDDR4 Memory 1866 MHz - 59.7 GB/s
Storage	32 GB eMMC 5.1
Power	7.5 W / 15 W

Table 4.1. NVIDIA Jetson TX2's Key Specs [25]

Jetson modules run Linux as an OS. NVIDIA provides JetPack SDK, which is a collection of software for Jetson modules. The main part is L4T (Linux For Tegra) which consists of flashing utilities, bootloader, Linux Kernel, NVIDIA drivers, and a sample filesystem based on Ubuntu.

4.1.3 Teensy 3.2

Teensy 3.2[3] is a development board with a single-core NXP Kinetis K20 MCU (MK20DX256VLH7), which is based on ARM Cortex-M4. It comes with a software library called Tennsyduino, which is an Arduino-compatible library. The following list summarizes some of the MCU's key parameters:

- ARM Cortex-M4 at 72 MHz
- 256K Flash, 64K RAM, 2K EEPROM
- USB device 12 Mbit/sec
- 34 digital input/output pins, 12 PWM output pins
- 21 analog input pins, 1 analog output pin, 12 capacitive sense pins
- 3 serial, 1 SPI, 2 I2C ports
- 1 I2S/TDM digital audio port
- 1 CAN bus
- 16 general purpose DMA channels

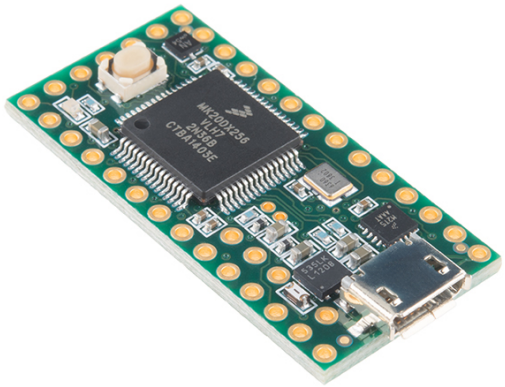

Figure 4.4. Teensy 3.2 Development Board [26]

4.1.4 VESC

VESC[4] is an open-source (both hardware and software) ESC created by Benjamin Vedder. It is based on a single-core ARM Cortex-M4 MCU which runs the open-source bldc firmware. There is also a sophisticated GUI tool called VESC Tool (formerly BLDC Tool) that allows configuration, firmware upgrades, and tuning of connected VESCs.

Throughout time, there have been many iterations of VESC's design, and different manufacturers have produced and sold VESC hardware. The VESC that is currently

[3] https://www.pjrc.com/store/teensy32.html
[4] https://vesc-project.com/

used on the CTU's F1/10 cars is Enertion FOCBOX VESC-X (with bldc firmware v2.18) which is very old and no longer manufactured as the company Enertion went of business. However, it still works. Compared to the latest VESC 6 MK5, it has fewer IO interfaces and fewer features in general.

4.1.5 LiDAR

Figure 4.5. Hokuyo UST-10LX LiDAR

4.1.6 IMU

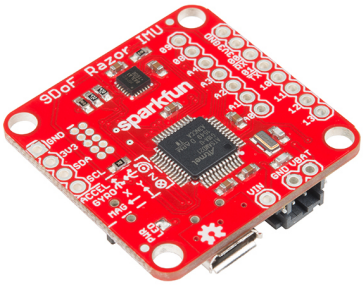

Figure 4.6. SparkFun 9DoF Razor IMU M0

4.1.7 Additional Components

The design requires a few additional components that were not mentioned in the previous sections. Most notably:

- a USB hub – Because NVIDIA Jetson TX2 features only one USB 3.0 port.
- An Ethernet USB adapter – In case we connect the LiDAR to the only Ethernet port on the Jetson, and we still want to use a wired network connection (might be useful during development).

4.2 Software Stack

The CTU F1/10 autonomous driving stack is based on ROS 1 Kinetic Kame. It consists of multiple components that can be used in various combinations to support different applications. a detailed overview of the CTU F1/10 platform architecture can be found

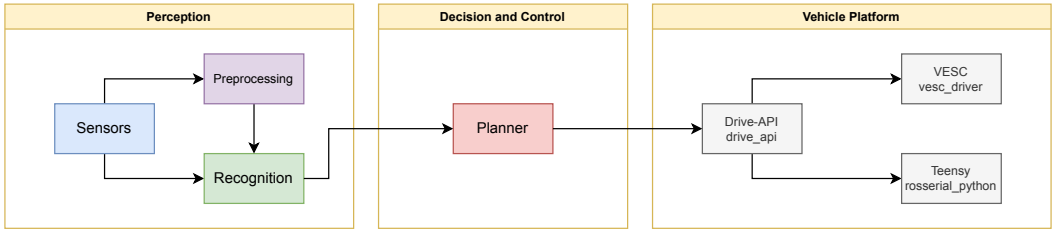

Figure 4.7. a high-level overview of a typical data flow is depicted in the the CTU F1/10 platform.

in Section 4.3.2 of [24]. a high-level overview of a typical data flow is depicted in Figure 4.7.

In the next subsections, we briefly describe some of the components that will be relevant for the migration.

4.2.1 Drive-API

Drive-API is a ROS node that provides a hardware-independent way of controlling the car. In fact, it provides several different methods for controlling the car (using different units). It loads the car-specific parameters from the ROS Parameter Server. Then it uses them to convert the car-agnostic control commands to car-specific Teensy and VESC control commands.

4.2.2 Simulation

In [24], the Stage simulator is introduced as an efficient and simple way to simulate F1/10 cars. ROS package `stage_ros`[5] provides the necessary bindings between Stage and ROS.

4.2.3 Teensy

In the CTU's F1/10 platform, the Teensy board has several responsibilities.

It decodes throttle and steering PWM signals from the RC receiver. It continuously sends the decoded duty cycles over USB to the Jetson so that it can be used by any high-level algorithms.

At the same time, it receives control commands (throttle and steering duty cycle) from the Jetson. Depending on the state of *emergency stop (eStop)*, it generates and outputs new PWM signals to control the servo and the VESC.

When eStop is active (manual override, signalized by a blinking orange LED), any control commands from the Jetson are ignored, and the data from the RC receiver are used instead.

When eStop is not active, then commands from the Jetson are used to control the servo and the VESC. Optionally the VESC can be controlled directly from Jetson using USB, which has the advantage of using eRPMs instead of just duty cycle. But when eStop is active, the VESC cannot be controlled using USB as the PWM throttle signal will simply overwrite the USB control commands.

Finally, using two manual switches, it is possible to bypass the Teensy board and use the RC receiver's PWM signals to control the servo and the VESC directly.

The communication between the Teensy and the ROS system running in the Jetson is implemented using `rosserial`[6]. This allows the firmware running in the Teensy to

[5] `https://wiki.ros.org/stage_ros`
[6] `https://wiki.ros.org/rosserial`

subscribe and publish to ROS topics as long as there is a special node (`rosserial` agent) running in the Jetson that handles the necessary translation between the ROS network and the USB.

4.2.4 NVIDIA Jetson TX2

ROS 1 Kinetic Kame targets Ubuntu 16. For that reason, the Jetson is flashed with NVIDIA JetPack 3.x, which is based on Ubuntu 16.

Chapter 5
Migration

Having introduced ROS 1 and 2 and having described the components of the CTU's F1/10 platform, we can move on to the actual migration from ROS 1 to ROS 2.

5.1 Note about Different ROS 2 Releases

Throughout the time we worked with the CTU's F1/10 platform, multiple ROS 2 versions were released.

- May 2020: 6th release ROS 2 Foxy (Ubuntu 20) (EOL: May 2023[1])
- May 2021: 7th release ROS 2 Galactic (Ubuntu 20) (EOL: November 2022)
- May 2022: 8th release ROS 2 Humble (Ubuntu 22) (EOL: May 2027)
- rolling release: ROS 2 Rolling

First, we targeted just Foxy and Galactic. Then, in order to get access to the latest features and performance improvements, we made the necessary changes to support the latest release, Humble, and the Rolling release as well. In the end, the migrated stack should be runnable in (at least) all these ROS 2 versions.

5.2 Scope of the Migration

Currently, the CTU's F1/10 platform is based on ROS 1 Kinetic Kame, which was released in May 2016 and deprecated in May 2021. That, among other things, means it runs on Ubuntu 16 and all ROS Python nodes are written in Python 2. The jump between ROS 1 Kinetic Kame and the current ROS 2 release might be significant. We might face a lot of silly issues with dependencies.

Because migrating all the code of the CTU's F1/10 platform at once would not be wise, we want to start with a self-contained part of it that we will actually try to migrate. Such part should meet at least the following criteria:

- It is possible to easily demonstrate its working in a simulator and on a physical car.
- It contains a minimal number of dependencies.
- It represents a typical autonomous driving application. That means reading data from sensor(s) (LiDAR), analyzing the data (perception), planning a trajectory (decision and control), and controlling the vehicle.

All these criteria are met by the Follow the Gap application, which is a part of the CTU F1/10 platform. It implements a reactive algorithm called Follow the Gap that was first introduced in [27]. It is a well-known part of the CTU's stack that was responsible for several wins of the CTU team in the F1Tenth Autonomous Competition in the past.

[1] Foxy was the first ROS 2 release with 3-year support. That signified the stability of this release. And Humble even came with a 5-year support plan.

Additionally, the migrated stack should be as compatible as possible (at least in the beginning) so that we can port more algorithms without rewriting their logic. This concerns especially the Drive-API.

Another thing to keep in mind is that the migration comprises not only porting the actual code to ROS 2, but also setting up the NVIDIA Jetson TX2 so that it can run ROS 2.

5.3 Follow the Gap Overview

We started by analyzing the operation of the Follow the Gap in its original ROS 1 environment. Then we also examined its code. We identified the packages that needed to be migrated to ROS 2.

There are three main parts that power the application:

- `perception/recognition/obstacle_substitution` – A package with one node that converts data from LiDAR to obstacles.
- `decision_and_control/follow_the_gap_v0` – A package that consists of two nodes. The first is the actual algorithm that looks for the biggest gap. Its output is the heading angle leading to the biggest gap. This heading angle is converted by the second node to the control commands, which are then sent to the Drive-API.
- Vehicle Platform (Drive-API, VESC, Teensy)

Apart from the nodes, there are messages' definitions, launch files, and various configuration files that need to be migrated too.

5.4 Process of the Migration

After we got familiar with the structure of the application in the ROS 1 environment, we actually proceeded with the migration itself. It was not an easy task as there were lots of tricky details and issues that needed to be solved. It required a lot of searching in the official documentation as well as examining the ROS 2 source code and examples as not all concepts were equally well documented.

In the following subsections, we describe some of the interesting challenges we faced during the process.

5.4.1 Parameters

Unlike ROS 1, ROS 2 has no global Parameter Server. Instead, in ROS 2, parameters are managed per node. Each node implements a parameter service that allows getting and setting the parameters during runtime. Initial values of parameters can be supplied at node startup time via command line arguments (parameters file). One of the most useful advantages this design brings is that all parameters can be easily changed during runtime, and nodes can implement custom behavior for handling the changes. In ROS 1, one must use solutions like `dynamic_reconfigure`[2] to get similar results.

One could argue that in some situations, it might be useful to have a "global parameter storage". Although that might be true sometimes[3], usually, the relationship between a parameter and a node where it should be defined is quite apparent.

[2] `https://wiki.ros.org/dynamic_reconfigure`

[3] Fortunately, ROS 2 provides an easy way of setting one or more the same parameters for multiple nodes using the asterisk (`*`) syntax in YAML parameter files.

For the CTU's F1/10 codebase, we were able to move all global parameters to logically-related nodes. At the same time, we got the ability to dynamically change the parameters, which proved to be useful, especially for the `follow_the_gap_v0_ride` node.

5.4.2 Teensy

As already mentioned in Section 4.2.3, the communication between the Teensy MCU and the Jetson is implemented using `rosserial`[4]. There is no direct port of `rosserial` to ROS 2 [28–29]. The official replacement in ROS 2 is micro-ROS [30], which is built around Micro XRCE-DDS middleware [31]. Given the scope and goal of the project (to bring as much of ROS 2 to MCUs as possible), the overall complexity and hardware requirements are rather high.

In order to avoid unnecessary complexity and overhead of micro-ROS, we decided to implement our own simple communication protocol between the Jetson and the Teensy. We implemented a C library (usable also from C++) for serializing and deserializing messages. This library can be used in the Teensy (bare-metal) as well as in the Jetson (Linux).

Next, we had to rewrite the firmware for the Teensy incorporating the new communication protocol. While doing that, we also tried to document some parts of the original code. However, there is still a big room for improvement. The new firmware implementation can be found in `pokusew/teensy-drive`[5] repository.

Finally, we implemented a ROS 2 node called `teensy_drive`[6] that connects to the Teensy via USB (CDC-ACM) and receives and send appropriate messages between Teensy and the ROS 2 network.

5.4.3 VESC

The `vesc`[7] package provides ROS 2 interface for VESC. However, we encountered a few bugs which motivated us to create a fork[8] fixing these bugs.

5.4.4 Stage simulator

In order to reuse the simulation from ROS 1, we needed to have working bindings between ROS 2 and the Stage simulator.

We found `stage_ros2`[9] package which is doing exactly that. However, it required a few changes in order to be usable in the latest ROS 2. We created a fork `pokusew/stage_ros2`[10] and implemented those changes. The author of the original repository even starred[11] our repository. Once there is more time, we will attempt to get our changes merged upstream.

Finally, we had to build both the Stage simulator (because there is no package available in recent Ubuntu releases) and our modified `stage_ros2` from the source. In the end, we integrated everything to the main workspace[12].

[4] https://github.com/ros-drivers/rosserial
[5] https://github.com/pokusew/teensy-drive
[6] https://github.com/pokusew/f1tenth-rewrite/tree/main/src/vehicle_platform/teensy_drive
[7] https://github.com/f1tenth/vesc/tree/ros2
[8] https://github.com/pokusew/vesc/tree/ros2-pokusew
[9] https://github.com/ymd-stella/stage_ros2
[10] https://github.com/pokusew/stage_ros2
[11] https://github.com/pokusew/stage_ros2/stargazers
[12] https://github.com/pokusew/f1tenth-rewrite

5.4.5 NVIDIA Jetson TX2 Setup

One of the most significant challenges was running the migrated stack on the Jetson. It included many different steps such as:

- Flashing an up-to-date OS image – NVIDIA's modified Linux – Linux for Tegra (L4T) together with all the drivers and other software components that make up the JetPack SDK.
- Creating a bootable SD card and making the Jetson boot from it instead of its internal eMMC. This way, we easily switch between different OS images, and we can also get bigger storage (the internal eMMC has a capacity of 32 GB).
- Configuring the system (udev rules, Wi-Fi, SSH, etc.) and installing any necessary software.

We collected a lot of documentation, notes, commands, links, and configuration regarding Jetson[13] in our `pokusew/ros-setup` repository.

Nevertheless, the main problem was Jetson's outdated software and its specifics. At the time of writing, the latest available NVIDIA JetPack is 4.6.1, which is based on Ubuntu 18. At the same time, the latest available Ubuntu LTS release is Ubuntu 22.

Officially, no ROS 2 version we target supports Ubuntu 18 (TODO: ref ROS 2 versions). However, one can attempt to build ROS 2 from sources even on older Ubuntus, but one must be prepared to face some problems with outdated system dependencies. We tried to do that for Foxy and Galactic and we were (after some struggles) successful.

Running ROS 2 natively on the Jetson is surely nice. Nevertheless, one has to build all application dependencies from their source code, as no prebuilt apt ROS packages are provided by ROS.

Another possibility is to use *Docker containers*. Containers (OCI containers) provide means for isolating applications running in the same OS. They are implemented using native Linux features, most notably namespaces and control groups. Containers are very efficient; they share the Linux kernel, thus having basically zero overhead. One could see containers as properly isolated OS processes.

Most importantly for us, we can create containers based on Ubuntu 20 or Ubuntu 22 filesystems and use them to run our ROS 2 stack even on the Jetson with Ubuntu 18. Docker containers can be configured to use the host network adapter, reducing any network overhead and simplifying the ROS 2 setup (it is exactly the same as if the application was running outside the container).

To sum up, we tried both options (build from source and Docker containers). While the Docker containers require more setup, they effectively solve the problem with outdated system dependencies in the systems like Ubuntu.

5.5 Result

Once we successfully migrated all the needed parts, we were able to demonstrate the working of the migrated application:

- **on the real F1/10 car**
- in the Stage simulator on Ubuntu and macOS[14]

The code can be found in `f1tenth_rewrite`[15] repository.

[13] `https://github.com/pokusew/ros-setup/tree/main/nvidia-jetson-tx2`

[14] Although it is possible to run ROS 2 on macOS, it requires a quite a bit of effort (everything must be built from source). We spent a lot of time making it work.

[15] `https://github.com/pokusew/f1tenth-rewrite`

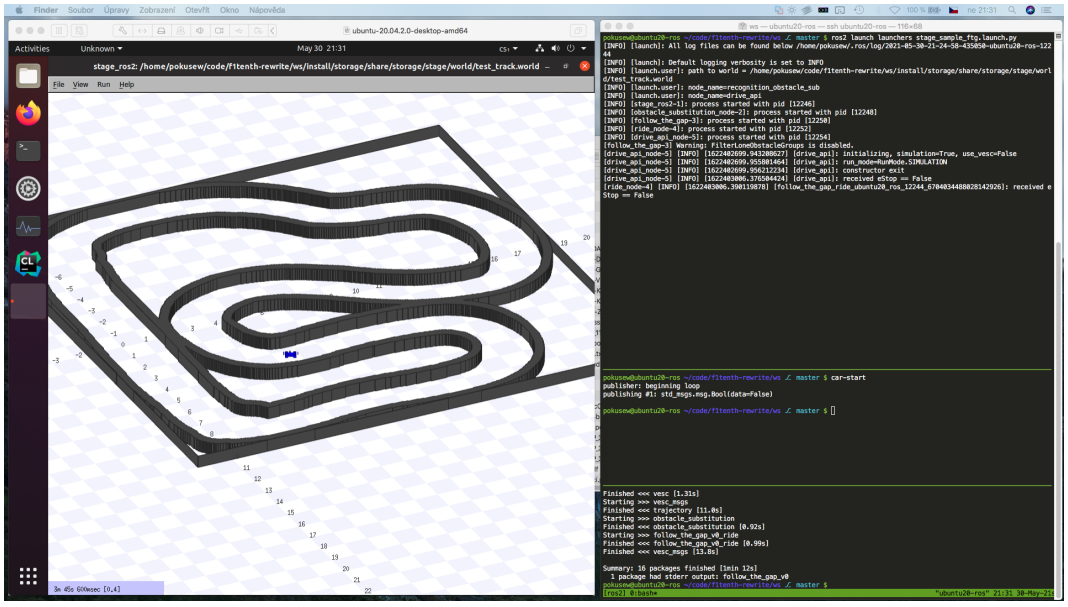

Figure 5.1. Running the Follow the Gap application in the Stage simulator in ROS 2 on Ubuntu 20.04 (as a VM on macOS)

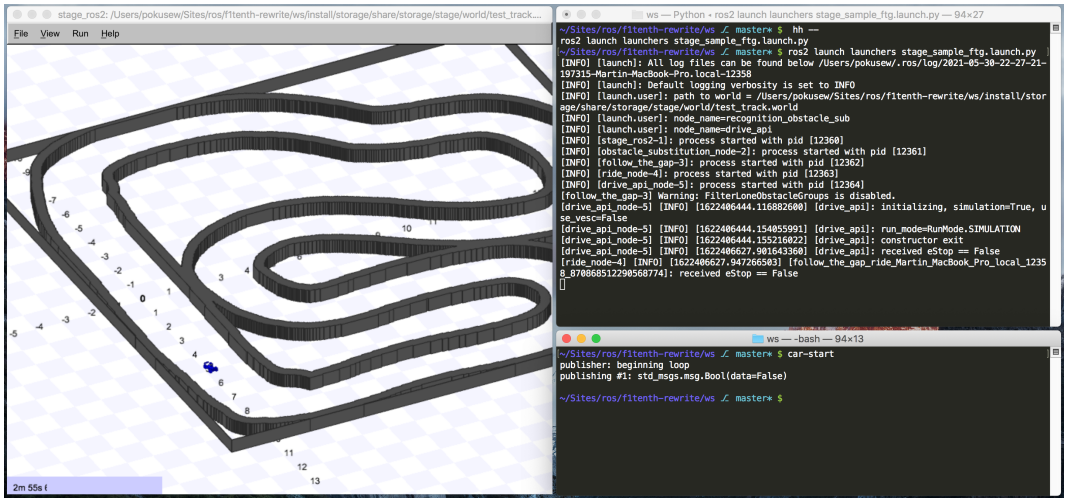

Figure 5.2. Running the Follow the Gap application in the Stage simulator in ROS 2 on macOS 10.14.6

Chapter 6
Evaluation and Experiments

After we migrated the stack, we tried to evaluate some of its properties. Especially, communication latencies (and the jitter) and execution times throughout the AV pipeline.

We used `ros2_tracing` framework. However, we decided to extend its analysis capabilities by integrating the work from [21]. This was necessary in order to for us to able to match the corresponding publication of messages with subscription invocations. After doing this matching we can compute communication latencies in the system correctly. Being able to reconstruct complete causal relationships in the system also enables us analyze application-specific end-to-end latency (time from perception to the corresponding control command).

Please refer to the `pokusew/ros2-tracing-experiments`[1] for the full description of the experiments we performed.

[1] `https://github.com/pokusew/ros2-tracing-experiments`

Chapter 7
Conclusion

In this thesis we dealt with the migration of the CTU's F1/10 autonomous driving stack from ROS 1 to ROS 2. First, we covered the necessary theory by describing both versions of ROS, `ros2_tracing` framework, and the components of the CTU's F1/10 stack. Then we migrated all the necessary parts so that we could demonstrate the Follow the Gap application running on ROS 2 on the real car and in the Stage simulator.

Then we spent a lot of time researching the ways to effectively evaluate runtime behavior of complex ROS 2 systems. We decided to use `ros2_tracing` and extend it with the message flow analysis based on the recent paper. We did not have enough time to document all our achievements in this thesis, but they are all publicly available online on GitHub[1].

Another result of our work is a publicly-available collection of setup guides, scripts, and documentation that covers various aspects of working with ROS. These guides have already helped several people.

We hope that the results of this thesis build a foundation that opens the way for the adoption of ROS 2 in CTU's F1/10 project.

[1] `https://github.com/pokusew/fel-bachelors-thesis`

References

[1] Evan Ackerman, and Erico Guizzo. *Wizards of ROS: Willow Garage and the Making of the Robot Operating System*. IEEE Spectrum, November 7, 2017. `https://spectrum.ieee.org/wizards-of-ros-willow-garage-and-the-making-of-the-robot-operating-system`.

[2] *Why ROS 2? – ROS 2 Design*. `https://design.ros2.org/articles/why_ros2.html`.

[3] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics*. 2022, 7 (66), eabm6074. DOI 10.1126/scirobotics.abm6074.

[4] *ROS Metrics*. `https://metrics.ros.org/`.

[5] *ROS Introduction – ROS Wiki*. `https://wiki.ros.org/ROS/Introduction`.

[6] *ROS 2 Documentation – ROS 2 Documentation: Rolling documentation*. `https://docs.ros.org/en/rolling/`.

[7] *Stats – ROS Index*. `https://index.ros.org/stats/`.

[8] *ROS Concepts – ROS Wiki*. `https://wiki.ros.org/ROS/Concepts`.

[9] *Distributions – ROS Wiki*. `https://wiki.ros.org/Distributions`.

[10] *Distributions — ROS 2 Documentation: Rolling documentation*. `https://docs.ros.org/en/rolling/Releases.html`.

[11] *ROS Client Libraries – ROS Wiki*. `https://wiki.ros.org/Client Libraries`.

[12] *ROS 1 Documentation – ROS Wiki*. `https://wiki.ros.org/`.

[13] *ROS Command-line tools – ROS Wiki*. `https://wiki.ros.org/ROS/CommandLineTools`.

[14] *roslaunch – ROS Wiki*. `https://wiki.ros.org/roslaunch`.

[15] *About internal ROS 2 interfaces – ROS 2 Documentation: Rolling documentation*. `https://docs.ros.org/en/rolling/Concepts/About-Internal-Interfaces.html`.

[16] *What is DDS?*. `https://www.dds-foundation.org/what-is-dds-3/`.

[17] *About different ROS 2 DDS/RTPS vendors – ROS 2 Documentation: Rolling documentation*.

https://docs.ros.org/en/rolling/Concepts/About-Different-Middleware-Vendors.html.

[18] *Launching/monitoring multiple nodes with Launch – ROS 2 Documentation: Rolling documentation.*
https://docs.ros.org/en/rolling/Tutorials/Intermediate/Launch/Launch-system.html.

[19] *ROS 2 Launch System.*
https://design.ros2.org/articles/roslaunch.html.

[20] Christophe Bédard, Ingo Lütkebohle, and Michel Dagenais. ros2_tracing: Multi-purpose Low-Overhead Framework for Real-Time Tracing of ROS 2. *IEEE Robotics and Automation Letters*. 2022, 7 (3), 6511–6518. DOI 10.1109/LRA.2022.3174346.

[21] Christophe Bédard, Pierre-Yves Lajoie, Giovanni Beltrame, and Michel Dagenais. Message Flow Analysis with Complex Causal Links for Distributed ROS 2 Systems. 2022,
http://arxiv.org/abs/2204.10208.

[22] Martin Vajnar. *Model car for the F1/10 autonomous car racing competition*. Master's Thesis, Czech Technical University in Prague, Faculty of Electrical Engineering. 2017.
https://dspace.cvut.cz/handle/10467/68472.

[23] Jan Dusil. *Slip detection for F1/10 model car*. Bachelor's Thesis, Czech Technical University in Prague, Faculty of Electrical Engineering. 2019.
https://dspace.cvut.cz/handle/10467/82910.

[24] Jaroslav Klapálek. *Dynamic obstacle avoidance for autonomous F1/10 car*. Master's Thesis, Czech Technical University in Prague, Faculty of Electrical Engineering. 2019.
https://dspace.cvut.cz/handle/10467/83424.

[25] *Jetson TX2 Module – NVIDIA Developer*.
https://developer.nvidia.com/embedded/jetson-tx2.

[26] *Teensy 3.2 Development Board image – micro-ROS.org*.
https://micro.ros.org/docs/overview/hardware/imgs/teensy32.jpg.

[27] Volkan Sezer, and Metin Gokasan. A Novel Obstacle Avoidance Algorithm: "Follow the Gap Method". *Robot. Auton. Syst.*. 2012, 60 (9), 1123–1134. DOI 10.1016/j.robot.2012.05.021.

[28] *Port rosserial to ROS 2 – Issue #365 – ros2/ros2 on GitHub*.
https://github.com/ros2/ros2/issues/365.

[29] *Notes on ROS2 and rosserial*.
https://newscrewdriver.com/2020/08/05/notes-on-ros2-and-rosserial.

[30] *micro-ROS – ROS 2 for microcontrollers*.
https://micro.ros.org/.

[31] *Micro XRCE-DDS compared to rosserial – micro-ROS*.
https://micro.ros.org/docs/concepts/middleware/rosserial/.

Appendix A
Glossary

- AV — autonomous vehicle(s), related to autonomous vehicles and autonomous driving
- BLDC — brushless DC electric motor
- CPU — central processing unit
- CTU — Czech Technical University in Prague
- DDS — Data Distribution Service
- eMMC — embedded MultiMediaCard
- ESC — electronic speed controller
- GPU — graphics processing unit
- HTTP — Hypertext Transfer Protocol
- IDE — Integrated Development Environment
- IDL — Interface Description Language or Interface Definition Language
- IMU — inertial measurement unit
- LiDAR — Light Detection And Ranging
- MCU — microcontroller unit
- OMG — Object Management Group
- OS — Operating System
- QoS — Quality of Service
- RC — radio-controlled / radio control
- RCP — Remote Procedure Call
- REP — ROS Enhancement Proposal, a document that standardizes certain aspect of ROS, a standard
- ROS — Robot Operating System
- SD card — Secure Digital card
- VESC — An opensource electronic speed controller, `https://vesc-project.com/`
- VM — Virtual Machine
- XML — Extensible Markup Language
- XML-RPC — an RPC protocol which uses XML to encode its calls and HTTP as a transport mechanism