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, I would like to thank 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

Tady bude abstrakt v českém jazyce.

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

/ Figures

Chapter 1
Introduction

Disclaimer: 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. They contain links to the GitHub repository with the implementation that will contain some background info. Additionally, 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 ROS – Robot Operating System – come in.

Since the Robot Operating System 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]. 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 [3]. Thus, now might be the right time to start migrating applications from ROS 1 to ROS 2 and benefit from the new possibilities.

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

Second, Chapter 2 explores tools and approaches that can be used to measure execution times, latencies, jitter, communication delays, and other parameters throughout

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

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 ?? 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 2, 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 1 and as we will elaborate in Chapter 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) offers a decent set of features, the CTU's F1/10 platform is, in fact, based on a now-prehistoric and (but wildly popular at the time) ROS 1 Kinetic Kame, which is now deprecated. Instead of migrating to just the latest ROS1, we can switch directly to ROS 2.
- 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" [4] 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" [5].

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 [6]. 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 so called "ROS Computation Graph" [7]. The nodes are represented by the graph's vertices, while the edges depicts the communication interactions.

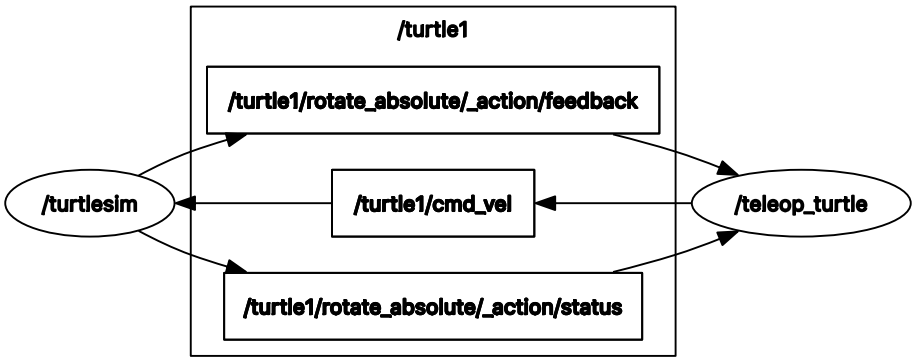

Figure 2.1. A simple ROS Computation Graph. Two nodes (represented as ellipses) (`/turtlesim` and `/teleop_turtle`) are running. The rectangles represent topics. The application is a part of the `turtlesim`[1] package.

[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. Service 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 preemptible task that has a goal, can provide continuous feedback and returns a result if it is not cancelled. 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** – 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 [7]. 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 [8]. ROS 1 distributions are listed at [8], ROS 2 distributions are listed at [9].
- **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" [7].
- **Package Resource Names** – a simplified method of "referring to files and data types on disk" [7]. 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 of a ROS node in a certain programming language. "It takes many of the ROS concepts and makes them accessible via code" [10]. It provides an API (functions, methods, classes) that allows to interact with other nodes (using pub/sub, services, actions

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

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. Version 1.0 was published in 2010. Since then, 13 ROS 1 distributions have been released [8]. 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 [11].

2.3.1 Architecture

In ROS 1 ROS Computation Graph, there always must 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 of connections connection 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 currently part of ROS Master (Parameter Server API). A key/value pair is called 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 transports 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 uses 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 covers all ROS 1 features. These include for example `rostopic`, `rosservice`, `rosparam`, `rosmsg`, `rosrun`, `roslaunch`, etc. [12].

[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 [13].

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." [5]

Full documentation of the latest ROS 2 release can be found at [14].

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

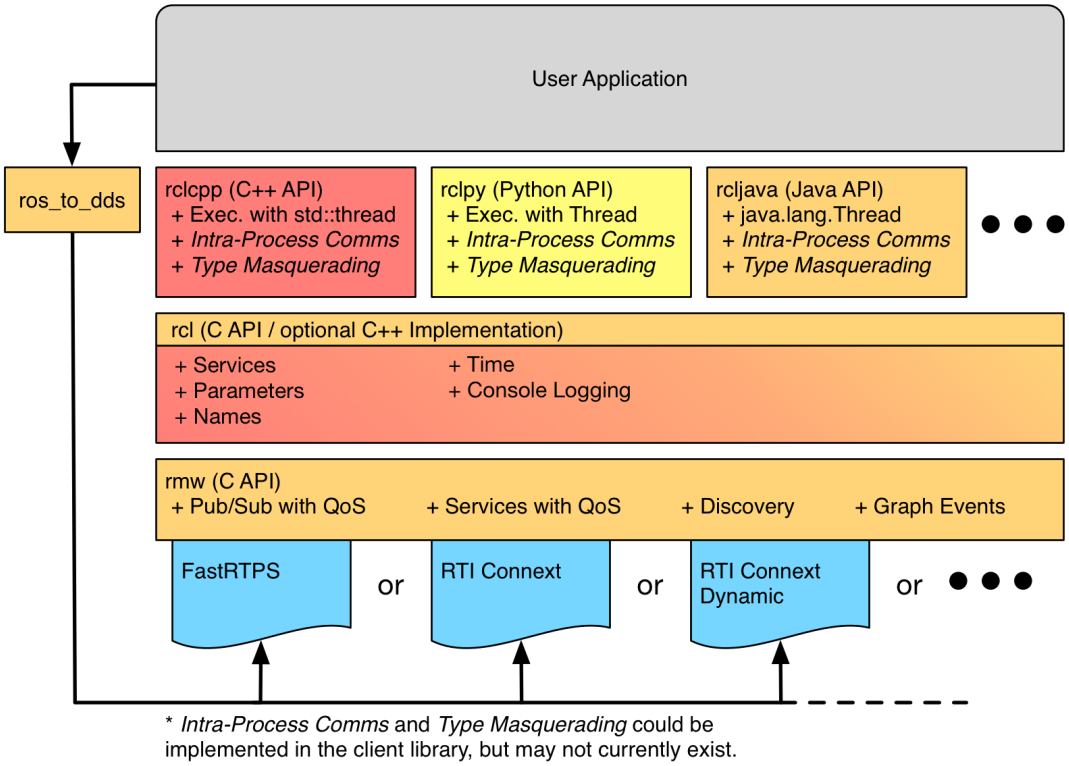

Figure 2.2. ROS 2 Architecture [15].

While ROS 1 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 *rcl* library that exposes a C API. Client Libraries then use *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], rcl library cannot directly communicate with the DDS implementation. Instead, there is an abstraction layer called *rmw* (ROS middleware) that provides a unified accesses to DDS. The specific DDS implementation can be even supplied dynamically at runtime.

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

2.4.2 Build System

ROS 2 uses ament as build system and colcon as 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 a 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` that has several subcommands.

2.4.4 Launch System

ROS 2 Launch System was redesigned. Currently, it uses Python 3 code to describe the system to launch [18–19].

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[6]. The following list summarizes the most notable differences between ROS 1 and ROS 2:

- Supported Platforms
  - *ROS 1:* Only Ubuntu officially supported.
  - *ROS 2:* Ubuntu, macOS, Windows officially supported. Yet, a lot of packages work only on Ubuntu.

[6] Although the actual Real-Time properties may differ dramatically based on various configuration and threading/execution model choice.

- Client Libraries
  - *ROS 1:* roscpp (C++03), rospy (Python 2, in later distributions Python 3)
  - *ROS 2:* rclcpp (C++14), rclpy (Python 3)
- Real-Time Support
  - *ROS 1:* no
  - *ROS 2:* yes
- Runtime Node Composition
  - *ROS 1:* No. But Nodelets can be used.
  - *ROS 2:* Yes. Composable Nodes.
- Parameters
  - *ROS 1:* global parameter server
  - *ROS 2:* parameters per node (no global parameter server), out-of-the-box dynamic_reconfigure-like features
- Launch System
  - *ROS 1:* XML-based
  - *ROS 2:* Python-based
- Transport
  - *ROS 1:* TCPROS or UDPROS
  - *ROS 2:* Handled by DDS which offers fine-grained QoS.
- Communication Primitives
  - *ROS 1:* pub/sub, services, actions (not natively, but via actionlib)
  - *ROS 2:* pub/sub, services, actions
- Threading and Execution Model
  - *ROS 1:* not much customizable
  - *ROS 2:* granular execution models, custom executors
- Build
  - *ROS 1:* catkin + catkin_make/catkin_make_isolated
  - *ROS 2:* ament + colcon
- IDL
  - *ROS 1:* `.msg`/`.srv`
  - *ROS 2:* `.msg`/`.srv`/`.action` + extended features such as constraints

Chapter 3
Methods for Evaluating ROS

Please refer to the `pokusew/ros2-tracing-experiments`[1]

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

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 F1/10 Autonomous Racing Competition (TODO: add ref or footnote). Thanks to its affordability and similarity to a real car, the platform can be easily used for the development, testing, and verification of autonomous driving systems and related algorithms.

At CTU, multiple F1/10 models have been created and used [20–22]. In this thesis, we focus on the latest iteration of the design represented by the models codenamed `tx2-auto-3` 4.1 and `tx2-auto-usa`This 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 Tomas Nagy's) internship at the University of Pennsylvania. 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 Third 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 following table/list provides a quick

sity of Pennsylvania. .

overview of those components, which are then described in detail in the next sections. The diagram (ref diagram) 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

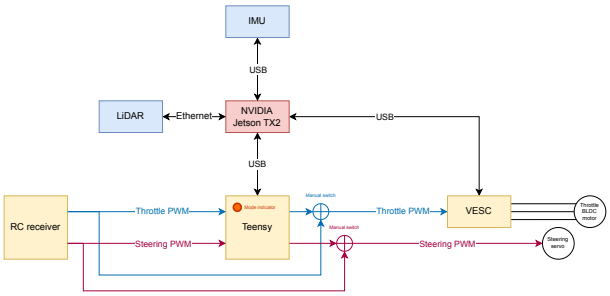

Figure 4.2. Overview of main components with depicted methods of communication. The yellow blocks are part of the low-level system, which always works independently of the main high-level system (NVIDIA Jetson TX2).

4.2 Software Stack

The software of stack is based on Ubuntu 16 (NVIDIA JetPack 3.x) and ROS 1 Kinetic Kame. The goal of this project is to migrate the stack to ROS 2 Foxy Fitzroy. The migration comprises not only of porting the actual code to ROS 2 (which is done in Chapter 5), but also of setting up the NVIDIA Jetson TX2 so that it can run ROS 2 (which is done in Chapter ??).

The CTU F1/10 platform 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 in the Section 4.3.2 of [22].

Because migrating all the code of CTU F1/10 platform at once would not be smart, we need to select some 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 the real model.
- It contains 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 all met by the *Follow the Gap* application which is a part of the CTU F1/10 platform. It implements the Follow the Gap algorithm that was introduced in [23].

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 Fitzroy (Ubuntu 20) (EOL: May 2023Foxy 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.
- May 2021: 7th release ROS 2 Galactic Geochelone (Ubuntu 20) (EOL: November 2022)
- May 2022: 8th release ROS 2 Humble Hawksbill (Ubuntu 22) (EOL: May 2027)
- rolling release: ROS 2 Rolling Ridley

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 (footnote: technical debt), 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.

5-year support plan.)

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 [23]. 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.

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 (which is done in Chapter ??).

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`[1] to get similar results.

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

One could argue that in some situations, it might be useful to have a "global parameter storage". Although that might be true sometimes (footnote: pass one param to all nodes), usually, the relationship between a parameter and a node where it should be defined is quite apparent.

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 (TODO: ref Teensy), the communication between the Teensy MCU and the Jetson is implemented using rosserial (TODO: https://github.com/ros-drivers/rosserial). There is no direct port of rosserial in ROS 2 [2] [3]. The official replacement in ROS 2 is micro-ROS ([4], [5]) which is built around Micro XRCE-DDS middleware. 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.

Finally, we implemented a ROS 2 node called `teensy_drive` 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 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`[6] 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`[7] and implemented those changes. The author of the original repository even starred (footnote) 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 (TODO: footnote).

5.4.4 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:

[2] https://github.com/ros2/ros2/issues/365
[3] https://newscrewdriver.com/2020/08/05/notes-on-ros2-and-rosserial
[4] https://micro.ros.org/
[5] https://micro.ros.org/docs/concepts/middleware/rosserial/
[6] https://github.com/ymd-stella/stage_ros2
[7] https://github.com/pokusew/stage_ros2

- 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 in our ros-setup repository (footnote: https://github.com/pokusew/ros-setup/tree/main/nvidia-jetson-tx2).

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 Jeston 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 car
- in the Stage simulator on Ubuntu and macOS (footnote: we also made it run on macOS)

The code can be found in `f1tenth_rewrite`[8] repository.

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

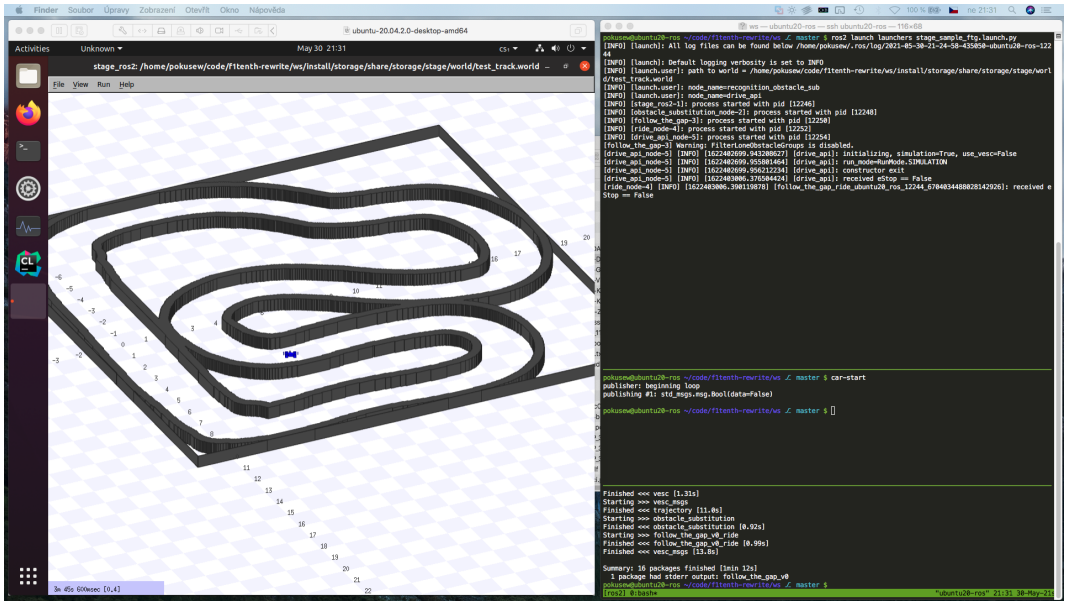

Figure 5.1. Running the Follow the Gap application in the Stage simulator in ROS 2 on Ubuntu 20.04 (as a VM on macOS)

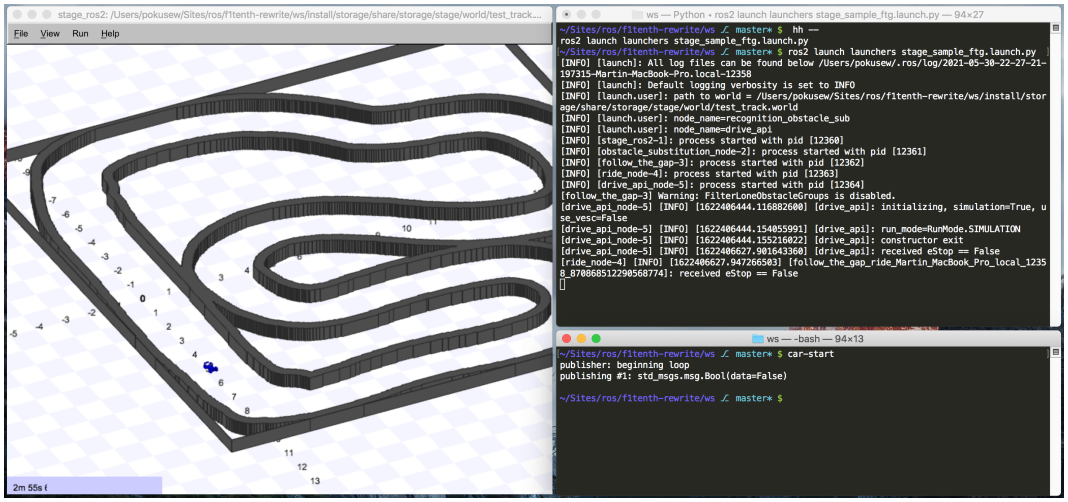

Figure 5.2. Running the Follow the Gap application in the Stage simulator in ROS 2 on macOS 10.14.6

Chapter 6
Evaluation and Experiments

Please refer to the `pokusew/ros2-tracing-experiments`[1]

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

Chapter 7
Conclusion

Please refer to the `pokusew/fel-bachelors-thesis/text/CONCLUSION.md`[1]

[1] `https://github.com/pokusew/fel-bachelors-thesis/blob/main/text/CONCLUSION.md`

References

[1] *History – ROS.org.*
https://www.ros.org/history/.

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

[3] *Users – ROS Metrics.*
https://metrics.ros.org/.

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

[5] *ROS 2 Documentation – ROS 2 Documentation: Foxy documentation.*
https://docs.ros.org/en/foxy/index.html.

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

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

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

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

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

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

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

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

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

[15] *About internal ROS 2 interfaces – ROS 2 Documentation: Foxy documentation.*
https://docs.ros.org/en/foxy/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: Foxy documentation.*
https://docs.ros.org/en/foxy/Concepts/About-Different-Middleware-Vendors.html.

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

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

[20] 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`.

[21] Jan Dusis. *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`.

[22] 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`.

[23] 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.

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