CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Cybernetics

Semestral Project

Migration of F1/10 Autonomous Driving Stack to ROS 2

Martin Endler
Open Informatics – Artificial Intelligence and Computer Science

May 2021

https://github.com/pokusew/fel-project

Supervisor: Ing. Michal Sojka, Ph.D.

Acknowledgement / Declaration

I would like to thank my supervisor Ing. Michal Sojka, Ph.D. for his guidance and support. Also, I would like to thank Ing. Jaroslav Klapálek for providing valuable information and advice regarding the F1/10 project.

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.

Abstract /

The new version of the Robot Operating System – ROS 2 – brings significant improvements over its predecessor ROS 1. We briefly describe both versions and compare their differences. Then we cover the process of migrating a ROS 1 application to ROS 2 on the example of the Follow the Gap app that is part of CTU's F1/10 project.

In the next part, we focus on setting up an NVIDIA Jetson TX2 module so that it can run ROS 2 applications and provide good developer experience in environments where Jetson module is shared among many students and researchers.

The first result of this project is a ROS 2 port of the Follow the Gap application. Its working is demonstrated in the Stage simulator on Ubuntu and macOS. The second result is a setup guide for NVIDIA Jetson TX2 that covers creating a fully-setup OS image with ROS 2 on a bootable SD card. The last result is a collection of setup guides and documentation that cover various aspects of working with ROS.

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

/ Contents

/ Figures

Chapter 1
Introduction

Since the Robot Operating System was started in 2007 [1], it has gained great popularity and has become the standard in robotics community. However, it has turned out that the original architecture (ROS 1) has some limitations concerning *performance, efficiency and real-time safety*. Thus, a new version of ROS – *ROS 2* – has been designed from ground up to allow more use-cases and solve 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 it might be the right time start migrating applications from ROS 1 to ROS 2 and benefit from the new possibilities.

The goal of this project is to migrate a selected part of the CTU's F1/10 project from ROS 1 Kinetic Kame to ROS 2 Foxy Fitzroy. The result should be a working port running on ROS 2 in a simulator and on a physical model car with NVIDIA Jetson computing module.

In the Chapter 2, both versions of ROS are briefly described while highlighting main differences of ROS 2 when compared to ROS 1.

In the Chapter 3, the software and hardware stack of CTU's F1/10 project is introduced. Then the Follow the Gap application is selected for migration to ROS 2. The migration is described in Chapter 4. The working of the migrated application is demonstrated in the Stage simulator on Ubuntu and macOS.

The Chapter 5 focuses on setting up an NVIDIA Jetson TX2 module so that it can run ROS 2 applications. The emphasis is given on providing a good developer experience in environments where the Jetson module is shared among many students and researchers. Thus, boot options of the Jetson modules are explored. Then a solution with a bootable SD card is presented.

Finally, **the achieved results** are summarized in the last Chapter 6.

Chapter 2
Robot Operating System

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.

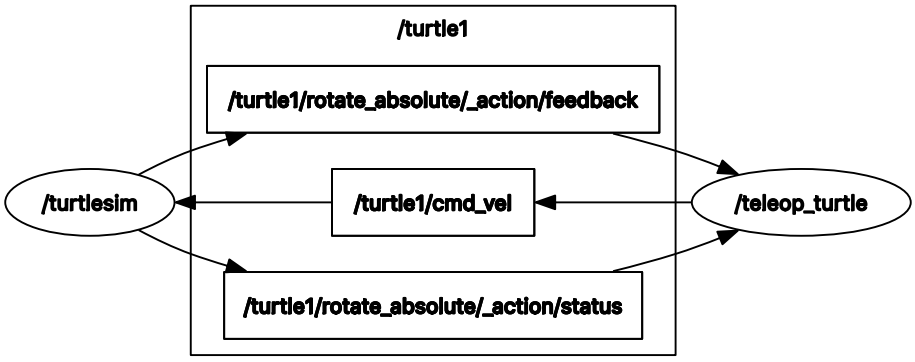

Figure 2.1. A simple ROS Computation Graph. Two nodes (represented as ellipses) (`/turtlesim` and `/teleop_turtle`) are running. The rectangles represent topics. The application is a part of the `turtlesim`[1] package.

[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. The version 1.0 was published in 2010. Since then, 13 ROS 1 distributions have been released[5]. The full list can be found at [8].

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[6]. 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].

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].

[5] As of 28th May 2020

[6] 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.

2.4 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

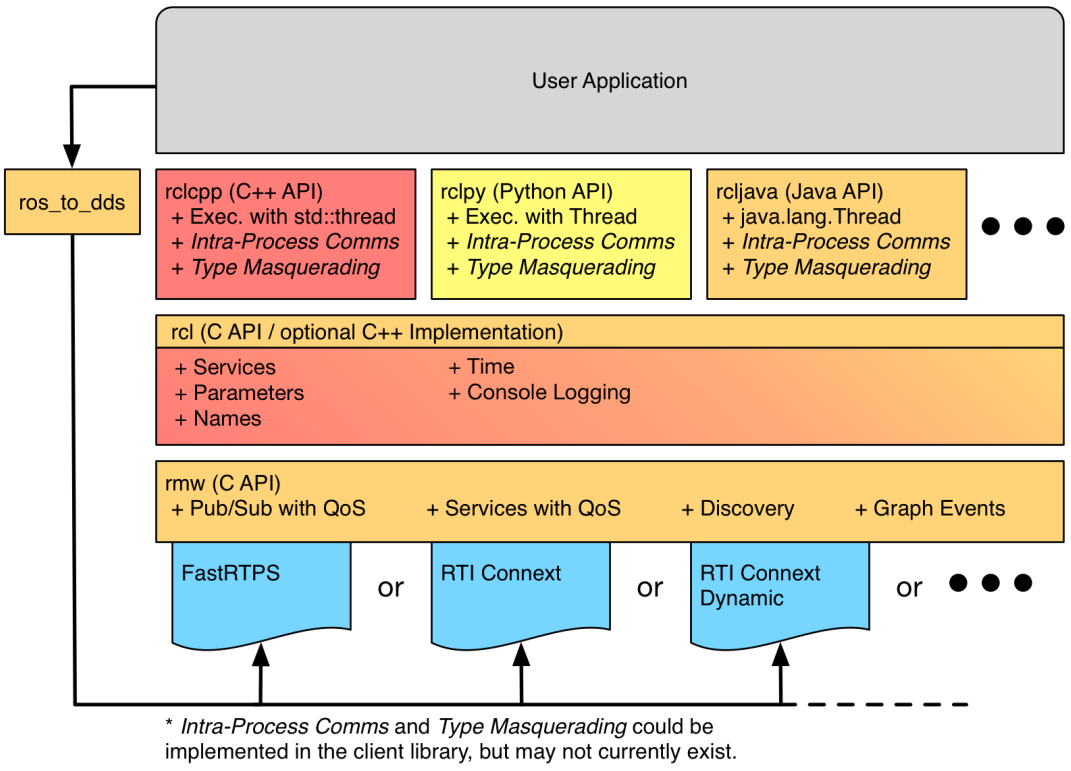

Figure 2.2. ROS 2 Architecture [15].

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[7]. 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.
- 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

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

- 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
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. Thanks to its affordability and similarity to a real car, the platform can be easily used for development, testing and verification of autonomous driving systems and related algorithms. At CTU, multiple F1/10 models have been created and used [20, 21, 22]. *Currently*, all of them are based on NVIDIA Jetson computing modules and powered by ROS 1 Kinetic Kame.

Throughout this project, we worked with the third model (`tx2-auto-3`) which is based on NVIDIA Jetson TX2. The model is shown in the Figure 3.1. The further descriptions of hardware stack are specific to this model.

Figure 3.1. The CTU's Third F1/10 Model Car based on NVIDIA Jetson TX2.

3.1 Hardware Stack

The most notable components (with respect to this project) are:

- NVIDIA Jetson TX2 Module – The central computing unit that runs the autonomous driving stack.
- Orbitty Carrier – A carrier board for the NVIDIA Jetson TX2 Module.
- VESC – Electric Speed Controller that control the engine.
- Teensy MCU – An auxiliary microcontroller for controlling the steering and handling the communication with an RC Transmitter (Manual Control).
- Hokuyo UST-10LX – LIDAR

3.2 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 4), but also of setting up the NVIDIA Jetson TX2 so that it can run ROS 2 (which is done in Chapter 5).

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 4
Migration of the Follow the Gap

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 (nodes) that power the application:

- `perception/recognition/obstacle_substitution` – It converts data from LIDAR to obstacles.
- `decision_and_control/follow_the_gap_v0` – It consists of two nodes. First plans the trajectory using the data about obstacles. While the second converts the plan to the control commands and sends them to the Drive-API.
- `vehicle_platform/drive_api` – It accepts control commands and controls the car accordingly (speed and steering).

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

After we got familiar with the structure of the application in 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 lof 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.

Nevertheless, we **succeeded** in the task and we **successfully migrated** all the code required to run the application in a simulator. The code can be found in `f1tenth_rewrite`[1] repository.

The most notable issue that has arisen is the fact that 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 features.

We benefited from the new parameters' possibilities in the `follow_the_gap_v0_ride` node where we got rid of the `dynamic_reconfigure`.

On the other hand, in some situations it might be useful to have a "global parameter storage". In fact, the CTU's F1/10 project uses this concept very much. However, even in ROS 2, we can implement a node that will act as a "global parameter storage" and will store the global car model configuration (such as sensors' types) so that the launch configurations can be shared among different car models.

[1] `https://github.com/pokusew/f1tenth-rewrite`
[2] `https://wiki.ros.org/dynamic_reconfigure`

4.1 Demonstration in the Stage simulator

To verify the working of the migrated application, we used the Stage simulator. The Stage simulator was introduced in [22]. We used `stage_ros2`[3] package as bridge between ROS 2 system and the Stage simulator. However, the installation of Stage simulator was not so straightforward as it was not present in package managers on Ubuntu or macOS. So we had to build it from sources. The exact steps were different for each platform. As a part of this project, a colcon workspace `ros2-build`[4] was created that (among other things) allows building and using the Stage simulator.

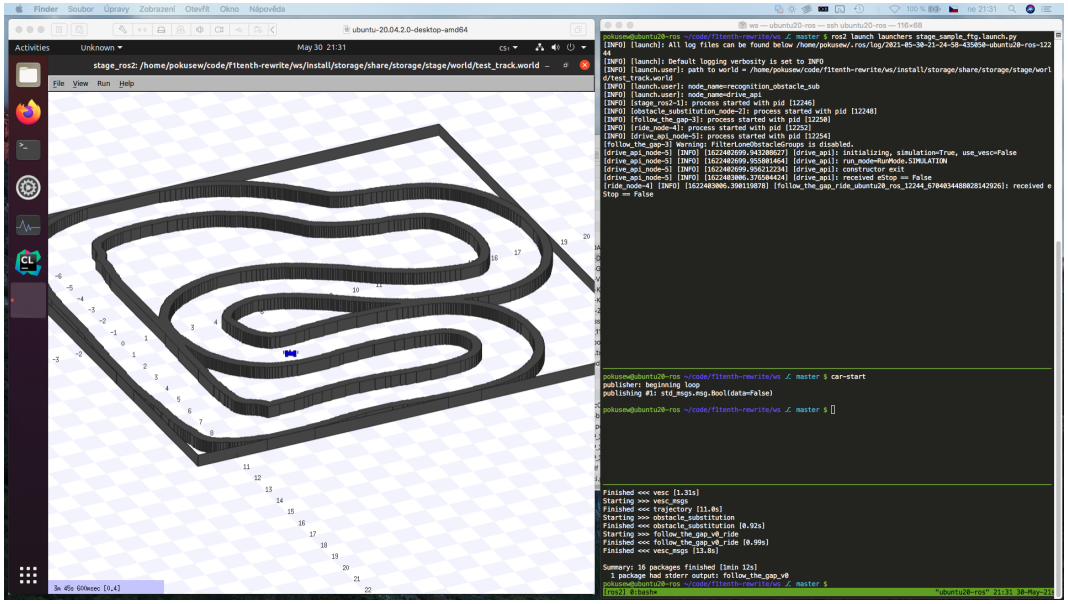

Figure 4.1. Running the Follow the Gap application in the Stage simulator in ROS 2 on Ubuntu 20.04 (as a VM on macOS)

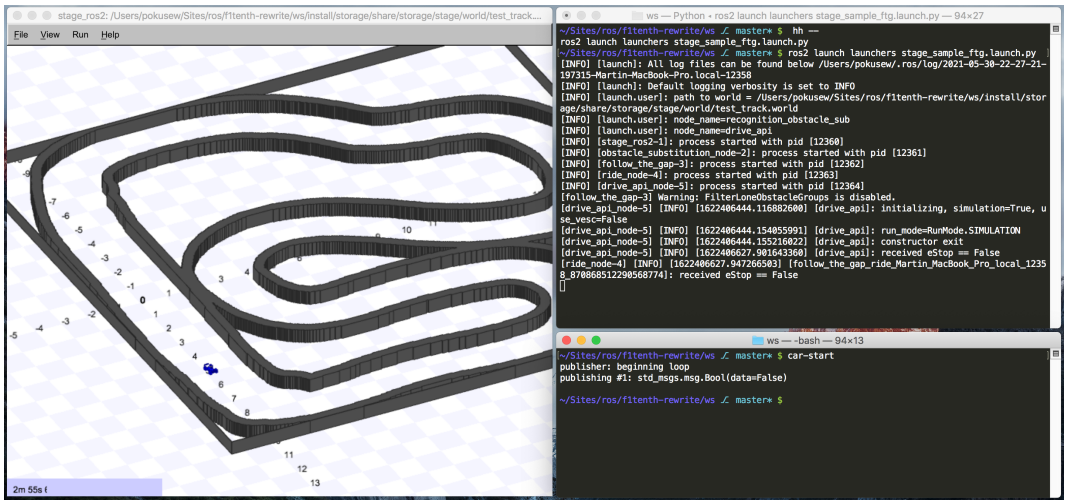

Figure 4.2. Running the Follow the Gap application in the Stage simulator in ROS 2 on macOS 10.14.6

[3] https://github.com/ymd-stella/stage_ros2
[4] https://github.com/pokusew/ros2-build

Chapter 5
NVIDIA Jetson TX2 Setup

In order to run ROS 2 on NVIDIA Jetson TX2, we needed to prepare an appropriate OS image. In this part of the project, we worked with the NVIDIA Jetson TX2 Module on the NVIDIA Developer Kit carrier board.

5.1 Initial Setup

NVIDIA provides *JetPack SDK* which is a collection of software for Jetson modules. The current version is 4.5.1. It contains *L4T (Linux For Tegra)* 32.5.1 which consists of flashing utilities, bootloader, Linux 4.9 Kernel, NVIDIA drivers and sample filesystem based on Ubuntu 18.04.

We used the NVIDIA SDK Manager to flash the most-up-to-date version of JetPack to our Jetson TX2 module.

5.2 Building ROS 2 from Sources

Then we followed with the customization of the Jetson TX2's Ubuntu 18.04 system. We had to built ROS 2 from sources as there are no binaries for Ubuntu 18.04 (only Ubuntu 20.04 is officially supported by ROS 2 Foxy Fitzroy).
The whole process is described at `https://github.com/pokusew/ros-setup/blob/master/nvidia-jetson-tx2/SETUP.md`.

5.3 Bootable SD Card

Normally, the OS of the Jetson module is installed on its internal eMMC memory. Sometimes, the size of the built-in eMMC memory (32 GB) is not enough. At other times, one may want to be able to easily switch between different operating systems and configurations. This is especially useful in environments, where one Jetson module is shared by many students and researches.

To support this use-case, Jetson TX2 can boot not only from its internal eMMC memory, but also from other media, including an SD card.

In fact, NVIDIA Jetson TX2 has a relatively complicated 3-level boot process which is in detail described at [24]. It is worth mentioning that all these three bootloaders and their configs are always stored on hidden partitions on the internal eMMC memory and cannot be moved anywhere else.

Fortunately, we are only interested in the last boot phase that is handled by U-Boot.

"The U-Boot includes a default booting scan sequence. It scans bootable devices in the following order:

- External SD card
- Internal eMMC
- USB device or NVMe device
- NFS network via DHCP/PXE

U-Boot looks for an `<rootfs>/boot/extlinux/extlinux.conf` configuration file of the bootable partition on each bootable device" [25].

The Jetson TX2 (resp. the NVIDIA Developer Kit carrier board) supports SD cards up to SDR104 mode (UHS-1).

We purchased a SanDisk MicroSDXC 64GB Extreme Pro (with an SD adapter) which is the best possible SD card (UHS-1) still supported by the Jetson TX2. We inserted the SD card into the Developer Kit SD card slot (while the Jetson TX2 was powered off as it does not support hot-plug). After powering the Jetson up, we formatted the card as `ext4` with a single partition. To this partition we copied the previously configured root filesystem from the internal eMMC memory. Then we updated the `/boot/extlinux/extlinux.conf` on the SD Card to incorporate new name of the root filesystem device that is passed from U-Boot to the Linux Kernel[1].

Then we rebooted the Jetson. **This time it successfully booted from the SD card.**

This configuration works well for the use-case introduced above. **The whole system runs from the SD card** (`extlinux.conf` and the Linux Kernel Image are both loaded from the SD card). This way each Jetson TX2 user can even customize the Linux Kernel Image for its application. When the SD card is not present during the boot, the Jetson boots normally from its internal eMMC memory.

One might be interested also in performance impact on the system when running from the SD card compared to the internal eMMC memory. We did not perform any performance impact measurements, but we did perform a *very simple*[2] read speed test to get at least a rough estimate. We got the following results:

- 246.6 MB/s (the internal eMMC memory)
- 86.5 MB/s (the SanDisk MicroSDXC 64GB Extreme Pro)

Please again note, that this test is mentioned just to get a rough estimate. A proper test (there might be big differences in sequential/random accesses, OS caches impact, etc.) should be done.

5.4 Running the Follow the Gap Demo

After we set up the ROS 2 environment on NVIDIA Jetson TX2, we wanted to demonstrate its working. Though normally one would not use GUI on NVIDIA Jetson TX2, in this part of project we did decide to try it. So we prepared also build of the Stage simulator for TX2's Ubuntu 18. Then we successfully ran the Follow the Gap application as can be seen in the Figure 5.1

[1] The resulting `extlinux.conf` can be found at `https://github.com/pokusew/ros-setup/blob/master/nvidia-jetson-tx2/boot-config/mmcblk2/extlinux.conf`

[2] During the test, a 10 MB of data was read 100 times.
The screenshot of the result for **the eMMC** is here:
`https://github.com/pokusew/ros-setup/blob/master/nvidia-jetson-tx2/boot-config/Screenshot%20from%202021-04-25%2018-16-28.png`
The screenshot of the result for **the SD card** is here:
`https://github.com/pokusew/ros-setup/blob/master/nvidia-jetson-tx2/boot-config/Screenshot%20from%202021-04-25%2019-45-06.png`

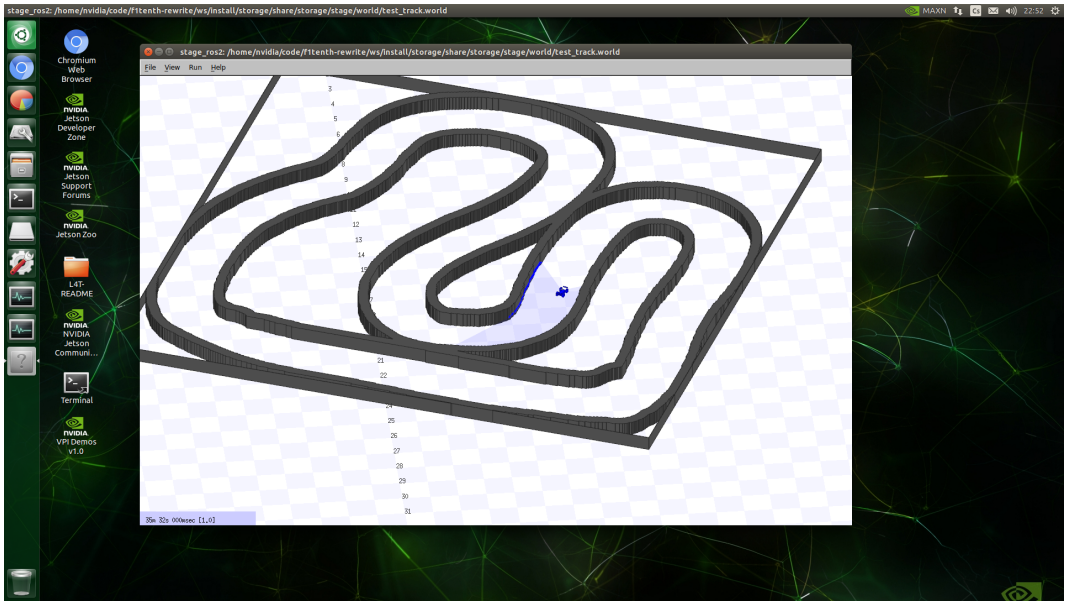

Figure 5.1. Running the Follow the Gap application in the Stage simulator in ROS 2 on NVIDIA Jetson TX2

Chapter 6
Conclusion

In this project we dealt with the migration of F1/10 autonomous driving stack from ROS 1 to ROS 2.

We described both versions of ROS and compared their differences. After introducing the CTU's F1/10 project, we selected the Follow the Gap application for ROS 2 migration. We successfully migrated it to ROS 2 and demonstrated its working in the Stage simulator on Ubuntu and macOS.

Then we covered the setup of NVIDIA Jetson TX2 for running ROS 2 applications. We also showed the Follow the Gap application running in Stage simulator on NVIDIA Jetson TX2. We presented the possibility of booting from SD card which is useful in environments where the Jetson module is shared among many students and researchers.

However, we did not run the Follow the Gap application on the real model car as the other steps consumed all the time allotted for this project. Fortunately, only a few additional (pretty straightforward) steps would be needed to make it work (namely Orbitty Carrier setup and testing of ROS 2 version of LIDAR and VESC packages).

During this project, a collection of setup guides and documentation was created. It covers various aspects of working with ROS, especially the usage of ROS with IDEs (CLion and Visual Studio Code) which may be very useful for other students.

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

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.

[24] *Jetson TX2 Boot Flow – NVIDIA Jetson Linux Developer Guide 32.5*.

[25] *U-Boot Customization – NVIDIA Jetson Linux Developer Guide 32.5*.

Appendix A
Glossary

- CPU — central processing unit
- CTU — Czech Technical University in Prague
- DDS — Data Distribution Service
- eMMC — embedded MultiMediaCard
- HTTP — Hypertext Transfer Protocol
- IDE — Integrated Development Environment
- IDL — Interface Description Language or Interface Definition Language
- LIDAR — Light Detection And Ranging
- MCU — microcontroller unit
- OMG — Object Management Group
- OS — Operating System
- QoS — Quality of Service
- 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