Declaration

I, Viplav Setia, born on 04.04.1995 in New Delhi, India, assure that I have done this work independently. All sources and references used for the completion of this thesis have been listed and cited accordingly. This thesis work was done in partial fulfillment of the requirements for the award of the degree of Master of Science in Mechatronics at Hochschule Ravensburg Weingarten and has not been used or submitted elsewhere for award of a degree, grade or in any publication.

Viplav Setia
Friedrichshafen, 31 January 2020

Acknowledgement

I would like to express my heartfelt gratitude to Prof Dr.-Ing Benedikt Reick and Prof Dr. rer. nat. Markus Pfeil for guiding me through the completion of my Master thesis and for their valuable suggestions.

I am extremely thankful to ALTEN GmbH and their colleagues who gave me this opportunity and the resources to do this thesis at their office branch in Friedrichshafen. They also supported me with their knowledge, expertise and created a pleasant working environment, without which it would have been difficult to move forward with this project.

Also, many thanks to my family and friends for their constant encouragement.

Abstract

The automotive industry is changing rapidly to new technologies like electromobility and automated driving. All major companies like Daimler, BMW, Tesla, Bosch, etc. are investing heavily to bring electric cars to the market and develop prototypes for automated driving. To support this change, middleware is required which is used as a means of data exchange between various sensors, control systems and actuators. The focus of this thesis is to test the new versions of the middleware, Robot Operating System(ROS), which offers support for embedded and real-time systems. Additionally, a model using the Gazebo robot simulator was developed to explore Advanced Driver Assistance Systems(ADAS) applications using a camera and a Light Detection and Ranging(LIDAR) sensor as an example to show the data transfer using ROS 2 for the automotive industry. To test the real-time performance of ROS2, an inverted pendulum demo was used and its simulation was visualized on a Linux system enabled with real-time capabilities. To test the version micro-ROS, a demonstrator was built using a STM32 microcontroller with a Nuttx Real-Time Operating System(RTOS) installed to show the data transfer of a pressure sensor. To test the real-time performance for this version, an algorithm was created to test the delay in data transfer with different data sizes. Finally, the results were analyzed and discussed which also helps in suggesting future research scope.

List of abbreviations, formulas and indexes

ADAS	Advanced Driver Assistance Systems
ANSI	American National Standards Institute
CPU	Central Processing Unit
DDS	Data Distribution Service
DDS-XRCE	DDS for eXtremely Resource Constrained Environments
I2C	Inter-Integrated Circuit
IP	Internet Protocol
LET	Logical Execution Time
LIDAR	Light Detection and Ranging
MCU	Microcontroller Unit
OFERA	Open Framework for Embedded Robot Applications
OpenCV	Open Computer Vision
POSIX	Portable Operating System Interface
QoS	Quality of Service
ROS	Robot Operating System
RTOS	Real-Time Operating System
UART	Universal Asynchronous Receiver-Transmitter
UDP	User Datagram Protocol
USB	Universal Serial Bus

Contents

1 Introduction

A modern car is a complex assembly of all kinds of sensors, control systems, actuators, drives and other mechanical components. A great amount of data is flowing between different components of a car which needs to be managed and also arrive at the right place at the right time. As shown in the figure below, Intel suggests about 4000 GB of data flow per day will take place in the future.

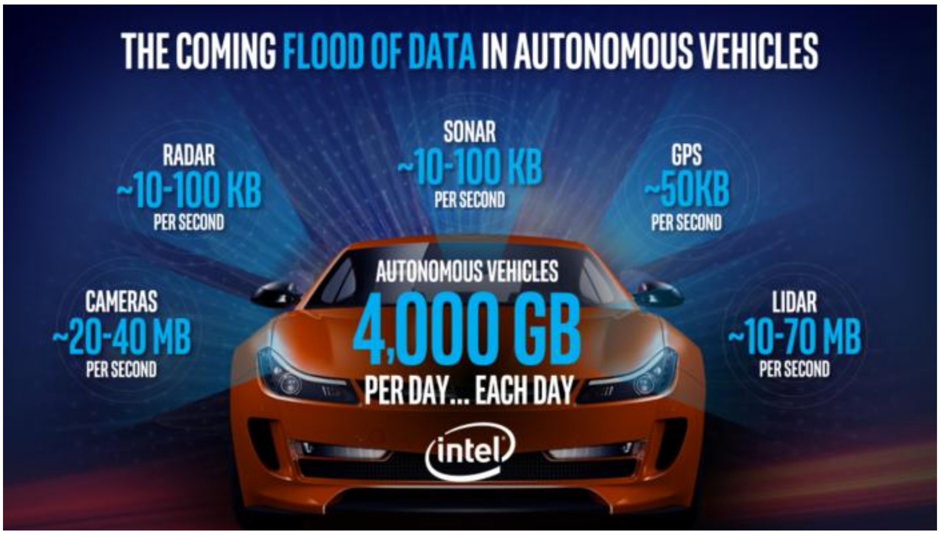

Figure 1.1: Data Stats in Autonomous Cars[1]

1.1 Motivation

For automotive applications, one major challenge is that all systems in the car should be real-time safe, that is, all systems of the car must give a guaranteed response within a specified time constraint. Missing a deadline can have disastrous consequences, such as, failure to apply the brakes at the right time after recognizing a person in front of the car may result in loss of life. One such software for communication data management is Robot Operating System(ROS). New versions of ROS, namely, ROS 2 and micro-ROS offer support for real-time systems and embedded boards. The goal of this thesis is to test the real-time capability and robustness of ROS 2 and micro-ROS under different test conditions.

1.2 Objectives

- Research on state of the art
- Apply ROS 2 concepts to explore Automotive ADAS Applications
- Set up STM32 microcontroller with RTOS and micro-ROS
- Test real-time performance of ROS2 using inverted pendulum demo
- Test real-time performance of micro-ROS
- Analyzing results and documentation

1.3 Robot Operating System(ROS)

ROS

The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.[2] It is an open-source software and is free to use for both research and commercial purposes.

But ROS does not guarantee deadlines and requires significant resources like high CPU usage, high memory consumption, etc. Therefore, ROS is not suitable for resource constrained real-time systems.

ROS 2

ROS 2 includes the components of ROS 1 which are great and improves those which are not. ROS 2 was developed to satisfy new use cases like real-time systems, embedded systems, non-ideal networks, production environments, etc. It also uses new technologies like Data Distribution Service(DDS). The software is developed and maintained by Open Robotics. It also offers support for different operating systems such as Linux, macOS, Microsoft Windows and different RTOSs.

ROS 2 Distributions

The ROS 2 Distributions are shown below in descending order of release date. Dashing Diademata is the first long term support version offered by the ROS developers. The work in this thesis is based on the version Crystal Clemmys as the Dashing version was released in May, 2019 and sufficient documentation for real-time testing was not available for it.

Figure 1.2: ROS 2 Distributions[3]

micro-ROS

micro-ROS puts ROS 2 onto microcontrollers, making them first class participants of the ROS 2 environment.[4] It uses a real-time operating system(RTOS), here Nuttx by default, and DDS for eXtremely Resource Constrained Environments(DDS-XRCE). In this thesis, ROS 2 Crystal version is used with Nuttx RTOS on a STM32 microcontroller which is a 32-bit microcontroller by STMelectronics. This project is funded by Open Framework for Embedded Robot Applications(OFERA) consortium consisting of Bosch, eProsima, Acutronic Robotics, etc.

Figure 1.3: micro-ROS Logo[5]

2 State of the Art

Real-time applications of ROS 2 have very recently come into the picture by the community. Many people have tested ROS 2 and have identified problems related to real-time performance. Also, the micro-ROS project is still in its infancy stage.
The core concepts of ROS 2, micro-ROS, embedded and real-time systems are mentioned in detail in this section. Also, the results of ROS 2 testing by some of the community members are stated.

2.1 ROS 2 Concepts

Node

An executable/application that runs a program/subprogram that communicate with each other via streaming topics is known as a node. It is used to communicate with other nodes using ROS client libraries which allow nodes to be written in different programming languages such as C, C++ and python. A robot may contain many nodes to control movement, analyse data, perform an operation like path planning, etc.
In ROS 2, discovery of nodes is automatic through the underlying middleware. Nodes advertise information to other nodes when they go online, offline and also periodically for new nodes to join and enable communication. ROS 2 design introduces node lifecycle, which helps to seperate real-time code path. All memory allocations are done during node initialisation.

Topic

Topics are named buses over which nodes exchange messages. Topics have anonymous publish/subscribe semantics, which decouples the production of information from its consumption. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic; nodes that generate data publish to the relevant topic. There can be multiple publishers and subscribers to a topic.[6]

Message

Nodes communicate with each other by publishing messages to topics. A message is a simple data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structs). msg files are simple text files for specifying the data structure of a message. These files are stored in the msg subdirectory of a package. Nodes can also exchange a request and response message as part of a ROS service call. These request and response messages are defined in srv files.[7]

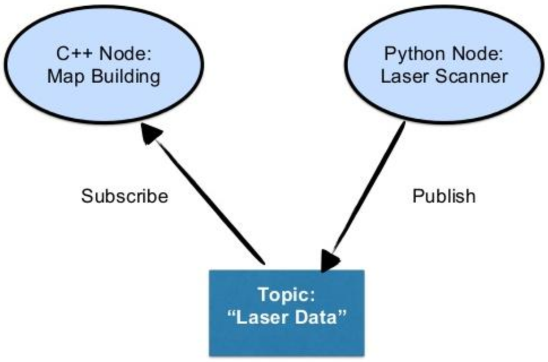

Figure 2.1: Working of Nodes, Topics and Messages[8]

Data Distribution Service(DDS)

Data Distribution Service(DDS) is a middleware standard which provides discovery, serialization and transportation to ensure dependable, high performance, interoperable, real-time data exchanges. In a distributed system, middleware is the software layer that lies between the operating system and applications. It enables the various components of a system to more easily communicate and share data. It simplifies the development of distributed systems by letting software developers focus on the specific purpose of their applications rather than the mechanics of passing information between applications and systems.[9]

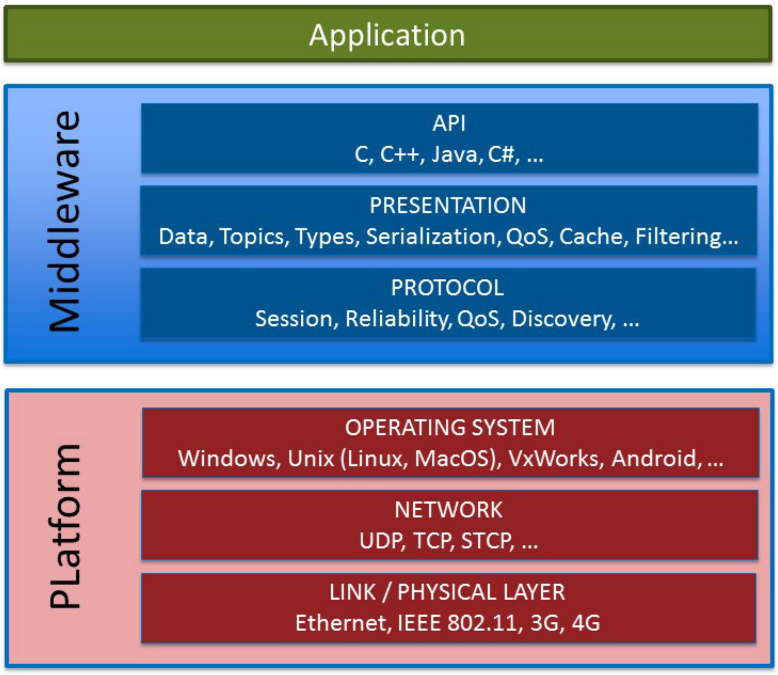

Figure 2.2: Software Layers in a Distributed System[9]

Quality of Service(QoS)

The data can also be shared with flexible Quality of Service (QoS) specifications including reliability, system health (liveliness), and even security. In a real system, not every other end-point needs every item in your local store. DDS is smart about sending just what it needs. If messages don't always reach their intended destinations, the middleware implements reliability where needed. When systems change, the middleware dynamically figures out where to send which data, and intelligently informs participants of the changes. If the total data size is huge, DDS intelligently filters and sends only the data each end-point really needs. When updates need to be fast, DDS sends multicast messages to update many remote applications at once. As data formats evolve, DDS keeps track of the versions used by various parts of the system and automatically translates. For security-critical applications, DDS controls access, enforces data flow paths, and encrypts data on-the-fly.[9]

The base QoS profile currently includes settings for the following policies:

- **History**
  - Keep last: only store up to N samples, configurable via the queue depth option.
  - Keep all: store all samples, subject to the configured resource limits of the underlying middleware.
- **Depth**
  - Size of the queue: only honored if used together with "keep last".
- **Reliability**
  - Best effort: attempt to deliver samples, but may lose them if the network is not robust.
  - Reliable: guarantee that samples are delivered, may retry multiple times.
- **Durability**
  - Transient local: the publisher becomes responsible for persisting samples for "late-joining" subscribers.
  - Volatile: no attempt is made to persist samples.[10]

ROS 2, by default, has QoS set to reliable, keep last history and volatile durability. In this thesis, only default QoS settings have been used as ROS 2 Crystal package was installed as a binary package which can be readily used. We can only modify these settings through configuration files of DDS middleware and then installing ROS 2 packages by source. ROS 2 Dashing provides an easier way of changing QoS settings in the source code of the application via Node Options package.

2.2 ROS 1 vs ROS 2

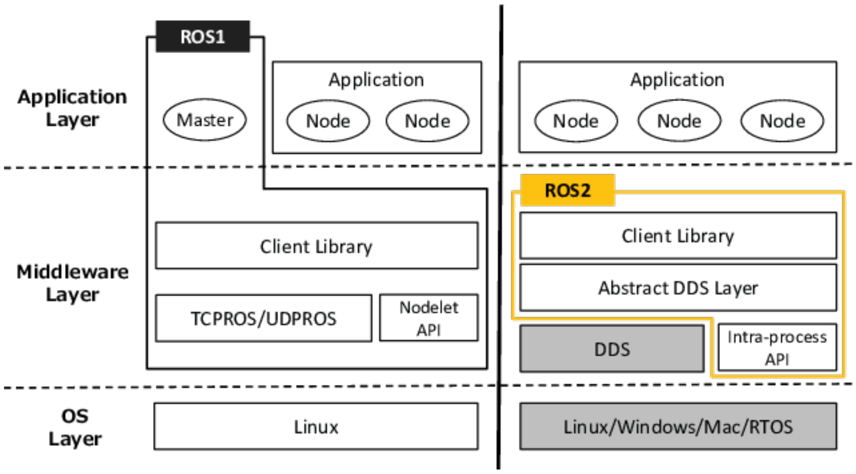

Figure 2.3: ROS 1 vs ROS 2 Architecture[11]

- **Application Layer**

  ROS 2 moves towards a distributed discovery mechanism where nodes advertise information to other nodes. ROS 1 has a centralized discovery mechanism where a master node is required to establish communication between nodes.

- **Middleware Layer**

  ROS 2 uses DDS standard through which discovery, QoS policies, serialization, and transport is provided which also offers real-time support. ROS 2 also requires new versions of client libraries like C++11 and C++14 and Python 3.5 at least. ROS 1 uses a custom transport protocol, centralized discovery, custom serialization format, and uses C++3 and Python 2 versions.

- **OS Layer**

  ROS 2 is supported on Linux, Windows 10, macOS and offers the possibilty to run it on a RTOS, whereas ROS 1 is only supported on Linux.

2.3 micro-ROS Architecture

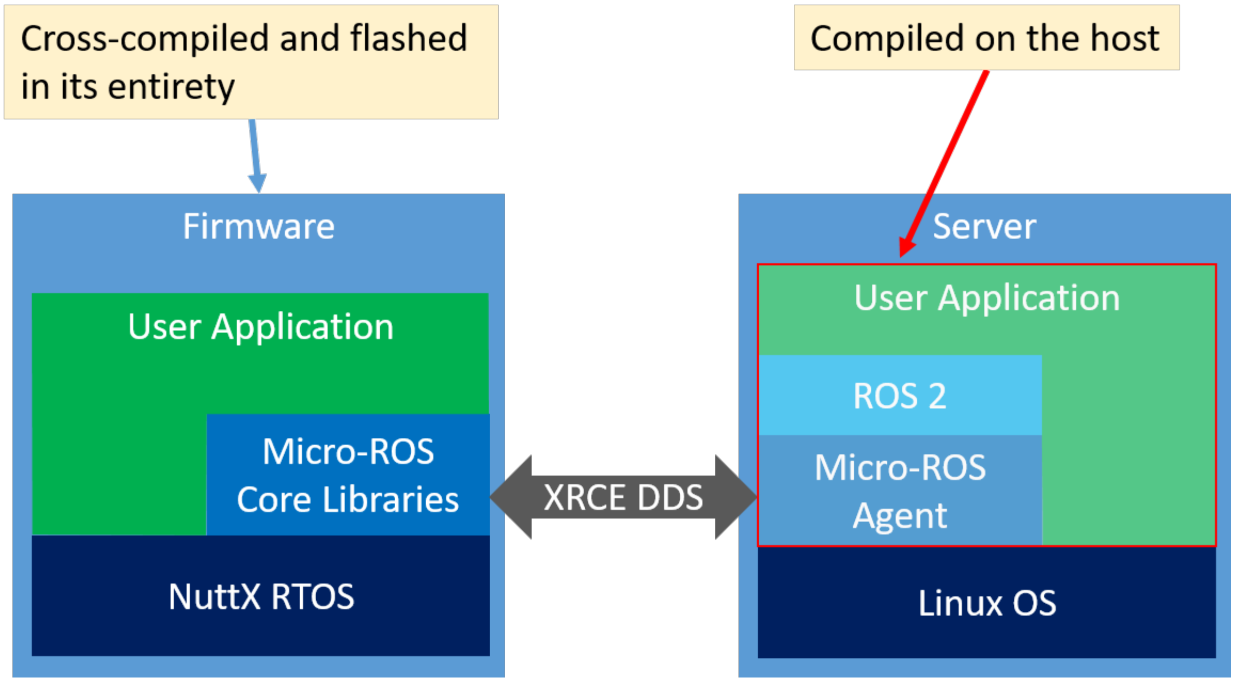

Figure 2.4: micro-ROS Architecture[12]

Firmware/Client

The firmware or client is the software cross-compiled(compiled on a different system than the target system) on the host - Linux and flashed onto the target - embedded microcontroller. The micro-ROS software uses Nuttx RTOS by default as its operating system. Nuttx can be configured to run different communication protocols and to run micro-ROS applications. Nuttx has a small footprint and is governed by the standards POSIX(Portable Operating System Interface) and ANSI(American National Standards Institute). After enabling micro-ROS and related communication settings in the Nuttx configuration, the user can run the micro-ROS nodes on the microcontroller which can be accessed by Linux through Universal Serial Bus(USB). The firmware communicates with the host through the DDS-XRCE and connects to the agent running on the host.

Agent/Server

The agent acts as a server for the clients and communicates with the microcontrollers. It runs on Linux and then can be used to connect with other ROS 2 nodes using the base ROS 2 versions.

Real-Time Executor

Robot applications require deterministic(predictable) execution of callbacks under all conditions and time constraints. Since the messages are buffered in DDS, ROS 2 introduces the concept of Executor,

to support execution management (prioritization of callbacks). The Logical Execution Time(LET) is a known concept in automotive domain to simplify synchronization in process scheduling. If refers to the concept to schedule multiple ready tasks in such a way, that first all input data is read for all tasks, and then all tasks are executed.[13]
This 2 step approach of the LET Executor guarantees a deterministic execution of callbacks.

2.4 Hardware and Communication Protocols Used

STM32 Microcontroller

A Microcontroller Unit(MCU) is a compact, integrated circuit which includes input/output peripherals, memory and a processor on a single chip. It is designed to govern a specific operation and is commonly found in automobiles, robots, mobile devices, vending machines, etc. In this thesis work, 32-bit MCUs have been used from the STM family of microcontroller boards. Two development boards, Olimex STM32-E407 and Waveshare STM32-Open407I-C, having similar architectures have been used for testing.

Figure 2.5: STM32 Development Boards[14][15]

Communication Protocols

Following are the communication protocols used in testing of micro-ROS:

- **Internet Protocol(IP)** - It is the main communication protocol to transmit datagrams across network boundaries through IP addresses. The MCU is assigned an IP address and a local area network is setup through ethernet wires between the computer and the MCUs to enable the micro-ROS Client Agent connection. User Datagram Protocol(UDP), part of IP suite, supported by micro-ROS is used in this thesis work.

- **USB-Serial** - PL2303 peripheral device is used to connect the board's UART(Universal Asynchronous Receiver-Transmitter) pins to the USB of the computer to enable communication via USB wire. However, the speed of data transfer is slower than IP because of the PL2303 driver used.

- **Inter-Integrated Circuit(I2C)** - is a multi-wire serial bus protocol to allow communication between small chips(slaves) with bigger chips(master). In this thesis work, a demonstrator for micro-ROS using a digital air pressure sensor was built which uses I2C protocol between the sensor and the MCU.

2.5 Real-Time Systems

Real-time systems should produce reproducable and correct computations at the correct time and be predictable even in a worst case scenario. Failure to respond within a set desired time(deadline) can cause damage to life or other resources. These systems have to be designed according to the dynamics of the physical process. They are often part of an embedded system and are employed in airplane sytems, automotive systems, satellites, power stations, and other critical applications. These systems are usually resource-constrained and still should offer low latencies. Jitter is the variation in the periodic loop time.

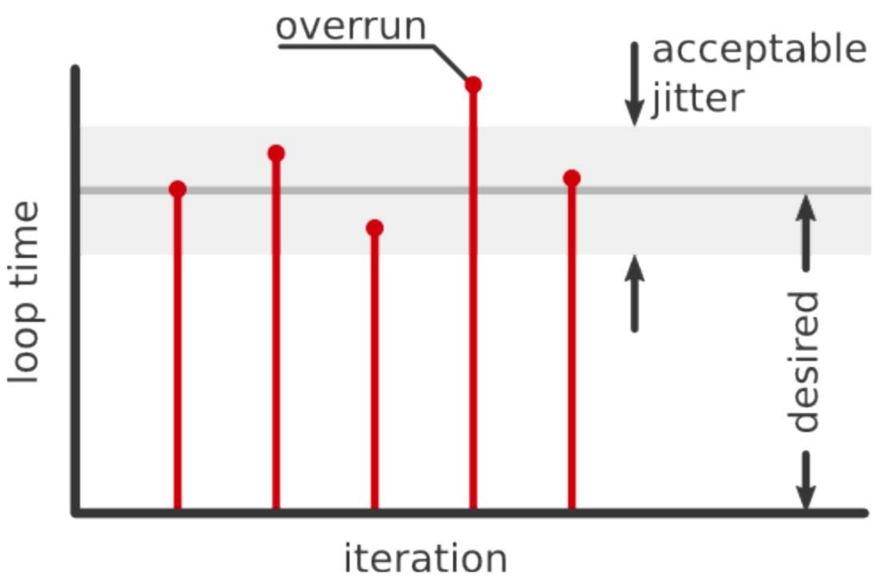

Figure 2.6: Deadline and Jitter in Real-Time Systems[16]

The different types of real-time systems are:

- **Soft** - The result can still be used after the deadline. For example, audio/video playback failure will only cause irritation to the user.

- **Firm** - The result does not have utility after the deadline. For example, in an automated manufacturing facility with high production rate, if the part does not arrive in time for a process to take place and the part is skipped or the machines stop, then it can cause financial losses to the facility.

- **Hard** - Missing a deadline can be catastrophic and can cause serious damage to life and property. For example, the landing gear of an airplane fails to deploy in time during landing.

2.6 Research by ROS Community

- **Inverted Pendulum demo** - The unstable arrangement of an inverted-pendulum is used as a means to test the real-time performance of ROS 2. A simulation is performed with ROS 2 where, the pendulum is balanced by a motor and a moving cart at the base. The motor command and sensor feedback are configured as ROS 2 messages and these are updated with a loop time of 1 msec. Linux preemtible kernel is used which offers real-time capability. High scheduling priority is given to the node and memory allocations are done during initialisation of the node. Dynamic memory allocations are blocked as they are not real-time safe. J. Kay and A.R. Tsouroukdissian have implemented this package with ROS 2 Alpha release. They also mention that DDS can be fine tuned for better real-time performance. Their goal was to get less than 3%(30 microsecs) jitter. Without any stress to the system, the maximum jitter was 3.51% , minimum was 0.16%, and mean was 0.46% but with stress on the processor, the maximum jitter was 25.8% , minimum was 0.14%, and mean was 0.38%. Also, 3 instances of overrun(loop time out of acceptable jitter range) were observed.

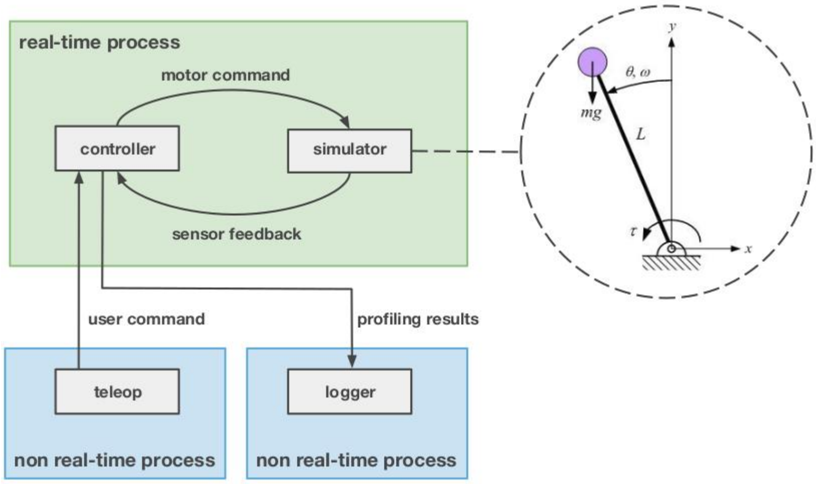

Figure 2.7: Inverted Pendulum Setup[16]

- **Non-deterministic scheduling by ROS 2 Standard Executor** - In a recent study by researchers at Bosch (for more details, see[17]), it has been found that the ROS 2 C++ Executor in version Crystal Clemmys has some undesired properties for real-time scheduling. They are:
  - The Executor gives highest priority to timers and therefore messages from the DDS queue are not processed in overloaded situations.
  - Non-preemptive round-robin scheduling of non-timer handles/entities(subscriptions, services,clients) leads to priority inversion, lower priority callbacks may block higher priority callbacks leading to high processing time. Also, this problem is further aggravated as only one message per handle is considered, even when multiple messages of the same topic are available, only one instance is processed by the Executor which causes backlog and hence, priority inversion.

These shortcomings led to the development of the LET Executor in micro-ROS to allow for deterministic execution.

- **High CPU Overhead by ROS 2 Executor** - In a study by Nobleo Technology (see [18]), it has been concluded that the ROS 2 SingleThreadedExecutor uses a lot of CPU power and generates overhead (unnecessary computation, memory allocations, etc.). ROS 2 Executor needs to be optimised otherwise normal ROS 2 cannot function properly on ARM A-class embedded boards. Based on the discussions between the ROS developers and the community, the Executor is undergoing some design changes to improve its performance. Some changes are planned for the next ROS 2 release Foxy due in May, 2020.

3 ADAS Applications using ROS 2

This is an open source project using License Apache 2.0 to understand simple ADAS applications using ROS 2 Crystal and Gazebo Simulator. You can drive around the robot in the simulator and have Lane Detection and Auto Brake when object is detected. To run this simulator, please refer to the instructions given in my Github repository, see link - ADAS Simulator using ROS 2 and Gazebo[19]

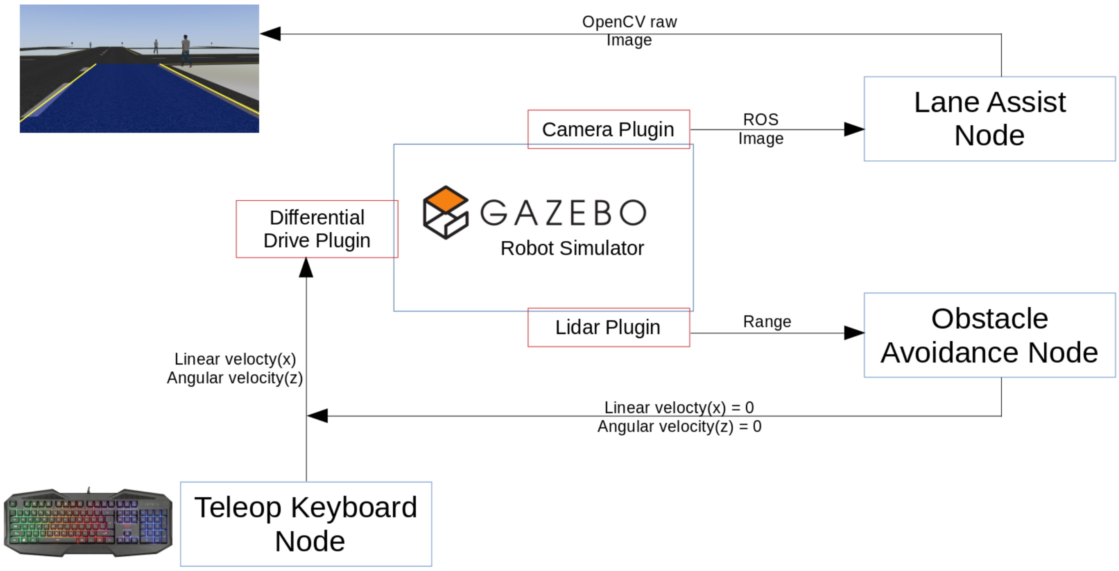

Figure 3.1: ADAS Simulator using ROS 2 and Gazebo

ROS 2 and Gazebo are both developed by Open Robotics. There are various plugins available to convert Gazebo to ROS 2 data. I have used a camera sensor and a laser sensor in the simulator with my robot which also has a differential drive and can be driven around with the help of a keyboard. The algorithms are developed using C++ language.
Firstly, a world file is created in the Gazebo Simulator to add roads and humans. The robot is modified from the differential drive demo already available in ROS 2 Gazebo tutorials and a camera and laser sensor is added. This setup consists of 3 nodes as explained below.

3.1 Lane Detection using Camera

Important steps to do lane detection for an image are :

- The input image subscribed from Gazebo simulator is a ROS message because of the camera plugin. First, it has to be encoded to a raw image to use Open Computer Vision(OpenCV) libraries. The raw image is then converted to a grayscale image.

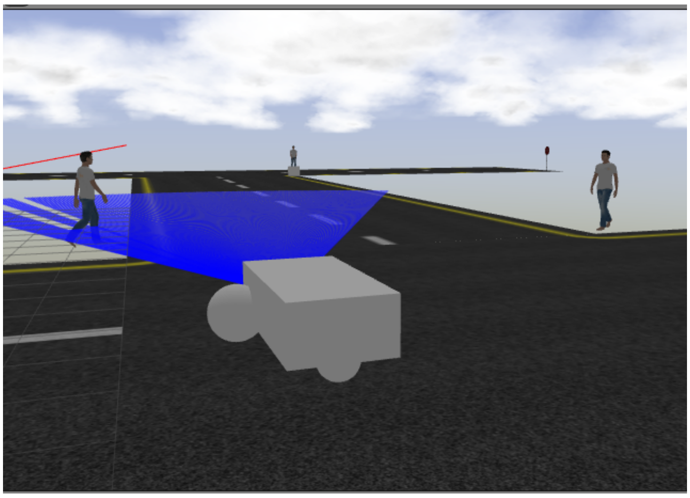

Figure 3.2: Gazebo Environment

- Canny Edge Detector from OpenCV libraries is applied to the greyscale image to detect edges.

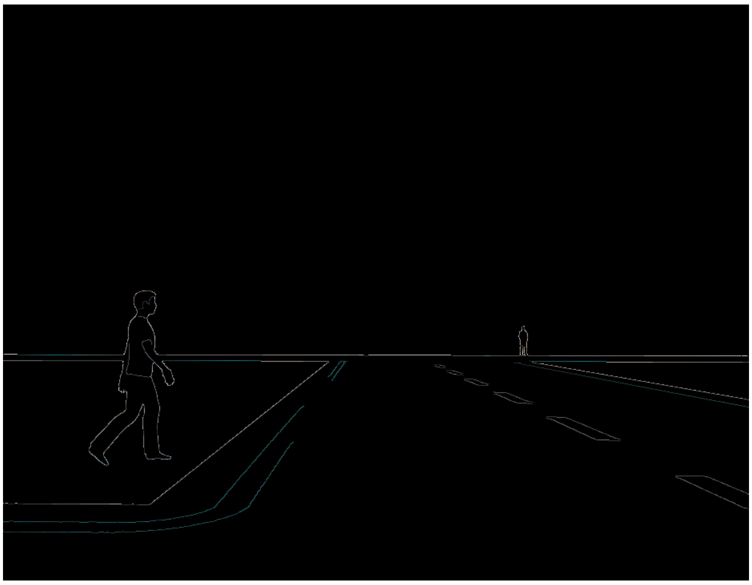

Figure 3.3: Canny Edge Detection

- An image mask is created according to the input image size which filters out the edges in the bottom half of the screen to show only the bottom edges. Then, Hough Transform from OpenCV libraries is applied to the resulting image which detects straight lines. The straight lines are then superimposed to the original raw image.

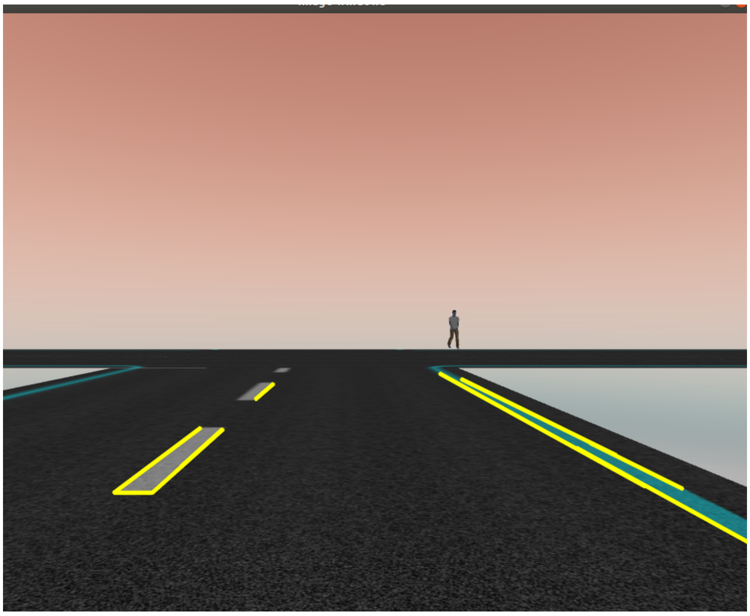

Figure 3.4: Line Detection using Hough Transform

- The lines detected are further optimised and a box is drawn connecting the two lines to display the lane. Also. turning advice to stay in the lane is printed onto the image calculating the direction in which the robot is heading.

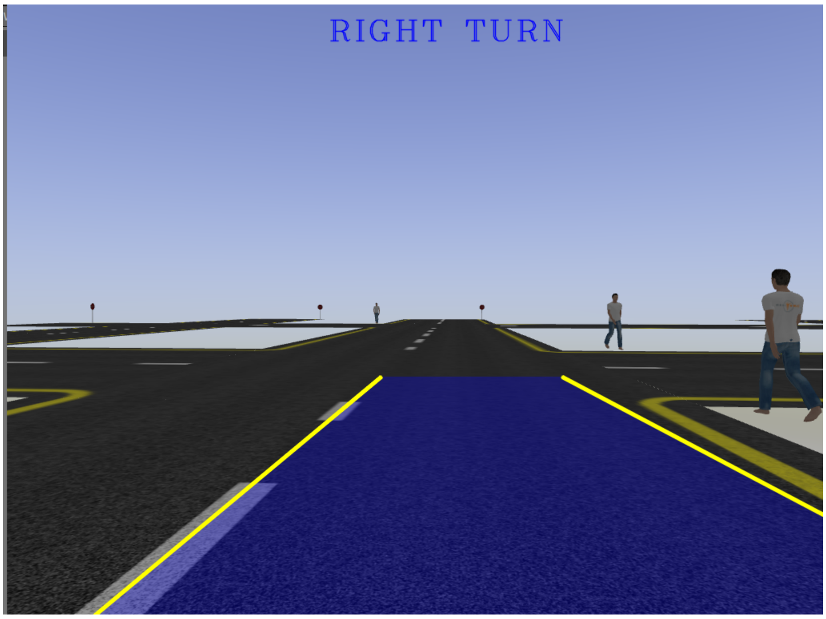

Figure 3.5: Lane Detection

3.2 Driver Control using Keyboard

The robot has 3 wheels and a differntial drive to enable turning. ROS 2 offers a built-in teleop_twist_keyboard package to send messages to the robot from the keyboard. The messages are already available in ROS 2 known as Twist messages. The differential drive plugin with Gazebo simulator can be enabled by modifying the robot model file in Gazebo.

3.3 Auto Stop using LIDAR

The laser sensor data gives out the minimum range of any object in its field. Range messages are already available in ROS 2. Then, an algorithm is applied to the incoming data and a condition is added. When the object is within a range of 4m, this node publishes a new Twist message to the robot telling it to stop. Also, it sends a "STOP, Reverse or Change direction" comment to the user and also prints out in the image running the in the lane detection node. The laser data can be visualized in the RViz package of ROS 2.

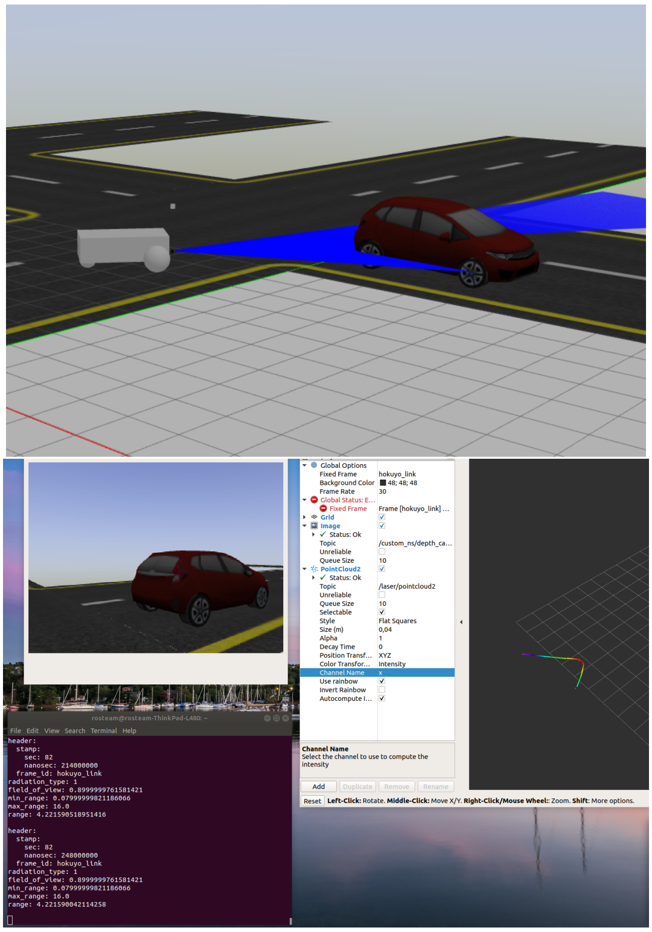

Figure 3.6: Laser and Camera Data Visualisation

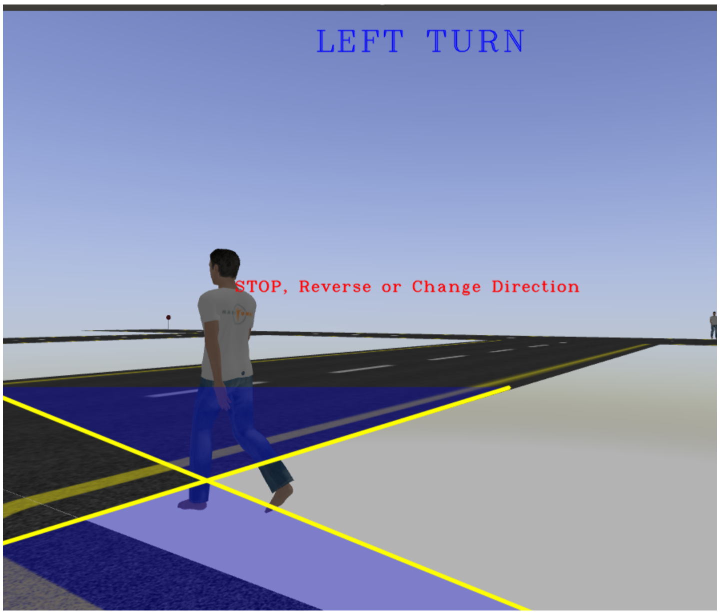

Figure 3.7: Auto Stop Feature

This project helps us to understand ROS 2 concepts and also explore its versatile libraries. Along with the Gazebo simulator, different kinds of robot applications can be tested and simulated in the worst case scenarios.
Different types of data can be accessed in a single node with multiple publishers or subscribers. Also, nodes can interact and discover themselves automatically, the user just has to source the ROS 2 software and launch the created nodes in different terminals in Linux.

4 Test Setup

4.1 Testing micro-ROS

4.1.1 Components

4.1.2 Procedure

4.2 Testing ROS2

4.2.1 Components

4.2.2 Procedure

5 Results

5.1 Latency Analysis in micro-ROS

node lifecycle partial

5.2 Latency Analysis in ROS 2

6 Conclusion and Future Scope

List of Figures

List of Tables

Bibliography

[1] B. Krzanich, "Data is the new oil in the future of automated driving." https://newsroom.intel.com/editorials/krzanich-the-future-of-automated-driving/#gs.kvj5y2/, 2016.

[2] O. Robotics, "About ros." https://www.ros.org/about-ros/, 27.12.2019.

[3] O. Robotics, "Distributions." https://index.ros.org/doc/ros2/Releases/#releases, 27.12.2019.

[4] micro ROS, "Overview." https://micro-ros.github.io/docs/overview/, 27.12.2019.

[5] micro ROS, "micro-ros." https://micro-ros.github.io/, 27.12.2019.

[6] O. Robotics, "Topics." http://wiki.ros.org/Topics, 27.12.2019.

[7] O. Robotics, "Messages." http://wiki.ros.org/Messages, 27.12.2019.

[8] A. Bail, "Robotics and ros." https://www.slideshare.net/ArnoldBail/robotics-and-ros, 27.12.2019.

[9] D. Foundation, "What is dds." https://www.dds-foundation.org/what-is-dds-3/, 03.01.2020.

[10] O. Robotics, "About quality of service settings." https://index.ros.org/doc/ros2/Concepts/About-Quality-of-Service-Settings/, 03.01.2020.

[11] T. A. Y. Maruyama, S. Kato, "Exploring the performance of ros2." Proceedings of the 13th International Conference on Embedded Software, pp. 5:1-5:10, 2016.

[12] micro ROS, "micro-ros-setup." https://github.com/micro-ROS/micro-ros-build/blob/crystal/micro_ros_setup/README.md, 03.01.2020.

[13] micro ROS, "Real-time executor." https://micro-ros.github.io/docs/concepts/client_library/real-time_executor/, 03.01.2020.

[14] Olimex, "Stm32-e407." https://www.olimex.com/Products/ARM/ST/STM32-E407/open-source-hardware, 03.01.2020.

[15] Waveshare, "Open407i-c standard, stm32f4 development board." https://www.waveshare.com/Open407I-C-Standard.htm, 03.01.2020.

[16] A. R. T. J. Kay, "Real-time control in ros and ros 2.0." ROSCon15, 2015.

[17] D. Casini, T. Blaß, 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)* (S. Quinton, ed.), vol. 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 6:1–6:23, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[18] Nobleo, "Ros2 performance." `https://github.com/nobleo/ros2_performance`, 2019.

[19] V. Setia, "Adas application with ros2 using camera and lidar using gazebo simulator." `https://github.com/Viplav04/ADAS_ROS2-OpenCV-Lane-Detection-Auto-Braking-with-LIDAR-Object-Detection-Gazebo-Teleo`, 2019.