Autonomous Robot for the Robotics Dojo Competition 2024

Mohamed I. Tuke, Obed G. Wambugu, Nasir J. Idris, Leonard M. Boma and Nathan KIngori (Limit Breakers)
Washington Kamadi, Micheal Kimani and Felix Wanyoike(Team 0804)

Abstract—**This paper presents the design and development of an autonomous robot intended for the Robotics Dojo Competition. The primary focus of the competition is to create a robot that can autonomously navigate a gamefield using a Lidar sensor as its main sensor for environmental perception. Each competing team was provided with a Raspberry Pi and encouraged to leverage ROS2 (Robot Operating System 2) for enhanced navigation and control. The robot employs real-time mapping to construct an accurate representation of the gamefield, enabling efficient and collision-free navigation. Critical elements of the design include the use of Lidar-based SLAM (Simultaneous Localization and Mapping) for environmental understanding, and ROS2 to manage sensor integration, path-planning algorithms for autonomous movement, and robust decision-making protocols. This paper details the key design decisions, system architecture, and algorithms implemented, along with a discussion on the challenges encountered and solutions devised.**

Keywords—**robotics, mobile robot, map generation, ROS, Linux.**

I. Introduction

In recent years, autonomous robots have gained significant traction across various domains, including industrial automation, search and rescue missions, and competitive robotics. The Robotics Dojo Competition presents a unique challenge that tasks participants with developing a fully autonomous robot capable of navigating a structured gamefield. To promote standardization and advanced development, each team was provided with a Raspberry Pi and encouraged to use ROS2 (Robot Operating System 2) as the foundation for robot control and navigation. The primary sensor for the competition is a Lidar unit, which enables the robot to perceive its environment and generate a real-time map for autonomous movement.

This paper describes the design and implementation of an autonomous robot built for this competition. The robot utilizes a Lidar sensor for environment perception and employs SLAM (Simultaneous Localization and Mapping) techniques to continuously update its understanding of the gamefield. The Raspberry Pi serves as the robot's computing platform, running ROS2 to integrate sensor data, manage navigation algorithms, and control movement. ROS2's flexibility and modularity are central to the robot's design, enabling efficient communication between components and facilitating the development of robust navigation strategies.

In addition to hardware considerations, this paper explores the software stack that enables autonomous decision-making, including path-planning algorithms and sensor fusion techniques. By combining the power of ROS2 with state-of-the-art mapping technologies, the robot is able to navigate the gamefield autonomously while avoiding obstacles and optimizing its movements. This paper outlines the design process, key decisions made, and the challenges encountered during the development phase, offering insights into the system's performance in a competitive environment.

A. Design Strategy

Our approach to the Robotics Dojo Competition is focused on achieving a balance between reliability and functionality while minimizing complexity to reduce potential failure points. Given the constraints of limited working hours and the technical requirements of the competition, the team prioritized the development of a robust and reliable navigation system using a Lidar sensor and ROS2 as the foundational software framework. The design philosophy revolves around building a solid core of autonomous navigation and mapping capabilities before introducing additional features that could complicate the system and impact overall reliability.

- Core Approach to Challenges

The primary challenge of the competition is to autonomously navigate a dynamic gamefield while mapping the environment in real time. Our strategy is to fully leverage the Lidar sensor to perform SLAM (Simultaneous Localization and Mapping), enabling the robot to construct a map of the gamefield while continuously localizing itself within that map. This approach ensures that the robot can avoid obstacles and plan efficient paths through the environment. By focusing on a well-tested and streamlined Lidar-based navigation system, we reduce complexity and improve reliability, ensuring that the core functionality of autonomous navigation is dependable under competition conditions.

- Addressing Tradeoffs: Capability vs. Robustness

Our team recognizes that increased sophistication in design often leads to additional points of failure. To manage this, we made a deliberate choice to focus on refining the capabilities of the robot's essential functions rather than adding complex secondary features that may not significantly enhance performance. Specifically, while advanced capabilities such as multi-sensor fusion or complex machine learning algorithms could potentially improve navigation or task execution, they also introduce greater computational demands and failure points. Instead, we chose to optimize the Lidar-based SLAM system and path-planning algorithms, focusing on tuning them to work reliably with the limited processing power of the Raspberry Pi.

By maintaining a streamlined design, the robot's overall reliability is enhanced. The use of ROS2 provides flexibility and modularity, which allows us to adapt to unforeseen challenges without needing to overhaul the entire system. This architecture supports the ability to add capabilities in future iterations without compromising the reliability of the existing system during the competition.

- Reliability vs. Complexity Tradeoffs

To further address the tradeoff between complexity and reliability, our design strategy includes a rigorous testing cycle. We allocated significant time toward testing and debugging core functionality, prioritizing stable and predictable behavior in real-world scenarios over implementing additional, unproven features. This decision ensures that the robot performs consistently and minimizes the likelihood of system failures during competition.

Furthermore, the simplicity of the hardware architecture reflects our desire to minimize mechanical and electrical failure points. By focusing on key components such as the Lidar sensor and Raspberry Pi, we reduced the risk of hardware malfunctions, which could be introduced by more complex or experimental setups. This emphasis on reliability was also extended to the software side, where we utilized ROS2's well-established libraries and community-tested modules to manage navigation, sensor data processing, and control algorithms. The use of established, well-documented frameworks contributes to the overall robustness of the system.

- Time Allocation: Capability vs. Testing

With the limited working hours available, our team prioritized iterative testing and refinement over adding new capabilities late in the design process. Each new feature or adjustment was subjected to multiple rounds of testing to ensure that it did not negatively affect the robot's core functionality. The team followed a methodical approach, first validating the Lidar-based mapping and navigation system in simulation before moving to real-world tests on the gamefield. This approach allowed us to identify and correct potential issues early, focusing our time on enhancing reliability and refining the system's core behaviors.

In conclusion, our design strategy for the Robotics Dojo Competition emphasizes the development of a robust, reliable, and simple navigation system using Lidar and ROS2. By prioritizing core capabilities and iterative testing, we reduced the complexity of the system while ensuring that it can perform consistently in a competitive environment. This balance between capability and robustness allows our team to focus on achieving dependable performance rather than chasing additional, potentially unreliable features.

B. *Vehicle Design*

The design of our autonomous robot for the Robotics Dojo Competition was a multidisciplinary effort, involving mechanical, electrical, and software components working in harmony. Each subsystem was chosen based on specific design methodologies, focusing on simplicity, reliability, and efficiency. Below is a detailed description of our approach to component selection, lessons learned from design iterations, and key software strategies that were employed.

The hardware structure of the robot consisted of appropriately selected electronic and mechanical components.

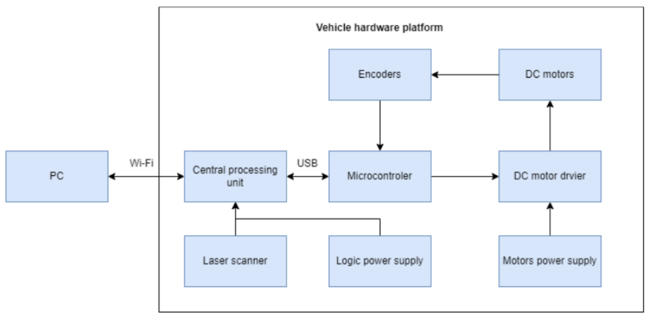

Fig.1. Hardware Structure of the Robot

- Mechanical Components

The mechanical structure of the robot is centered around a lightweight, durable acrylic chassis. A round chassis design was chosen to improve the robot's maneuverability, particularly in environments where tight turns and complex navigation are required. Acrylic was selected due to its balance between strength and weight, ensuring the robot remains agile while offering sufficient protection for internal components. Fig

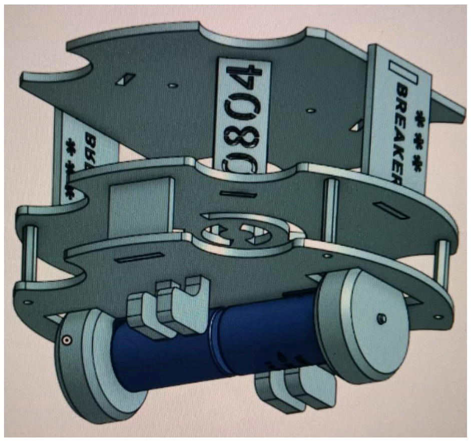

Fig. 2. CAD Model of the robot

Furthermore, 65mm diameter wheels Fig. 3.were chosen to provide sufficient ground clearance coupled with two caster wheels positioned at the front and back, allowing the robot to handle small obstacles without sacrificing speed or stability. Motor brackets and couplings were carefully selected to ensure proper alignment and secure attachment of the DC motors to the chassis.

Fig. 3. Wheel (65mm Diameter) and encoded motor

- Electrical Components

For the electrical components, two 12V 200 RPM DC motors with encoders were used to provide precise control over wheel movement. The use of encoders allowed for accurate feedback, which was crucial for ensuring consistent motion and integration with the navigation algorithms. The DC motors driving the platform were controlled by a two-channel L298N controller based on a double H-bridge. The controller achieved current efficiency on each of the channels equal to 2A. Additionally, the maximum voltage supplying the motors was 35V. This controller was protected against overheating by an added heat sink.

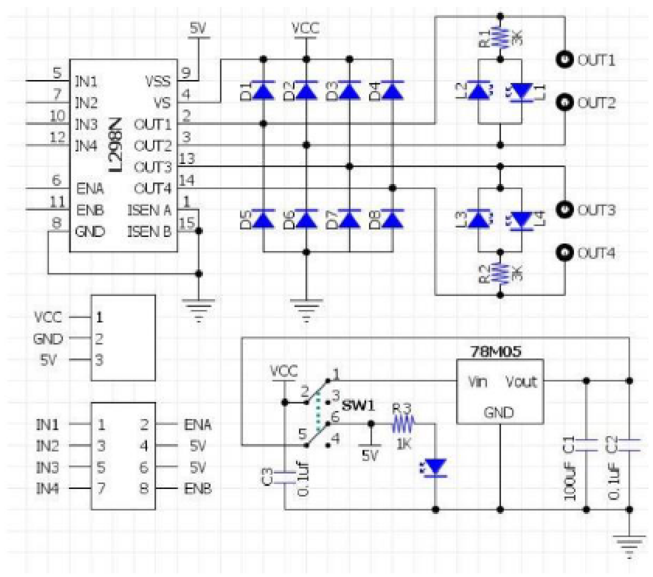

Fig. 2. Schematic diagram of the L298N DC motor controller.

The heart of the robot was a single-board Raspberry Pi 4B computer with a Cortex-A72 64-bit processor acting as the robot's on-board computer and the supporting Arduino MEGA (ATmega2560) microcontroller responsible for motor control. The on-board computer was the unit responsible for processing any data coming from the sensors and for wireless data transmission to the operator's personal computer.

The microcontroller, which was connected to the central unit via a USB connector, was responsible for receiving signals from the encoders and developing control commands transmitted directly to the motors via the GPIO pins. The purpose of using the Arduino MEGA microcontroller (Table 3) was to relieve the on-board computer in order to handle four system interrupts necessary for correct control (reading data from the encoders).

Table 2

RASPBERRY PI 4B SPECIFICATIONS

Parameter	Value
Processor	Broadcom BCM2711, Cortex-A72 64-bit SoC, ARM8v-A, 4 x 1.5 GHz
RAM	4 GB LPDDR4-3200 SDRAM
Memory	Flash: micro SD card slot
Supply voltage	5.1 V, 3 A via USB C
Wi-Fi interface	2.4 GHz and 5.0 GHz IEEE 802.11ac
Communication interfaces	UART, SPI, I2C, GPIO, DSI, CSI, USB 2.0, USB 3.0

Table 3.

ARDUINO MEGA MICROCONTROLLER SPECIFICATIONS

Parameter	Value
Microcontroller	ATmega2560
Clock frequency	16MHz
Flash Memory	256 kB
SRAM	8 kB
EEPROM	4 kB
Analogue Input Pins	16
Digital I/O Pins	54 (15 PWM outputs)
Supply Voltage	7 - 12V

The main source of information obtained from the surroundings was the RPLidar A1 laser scanner by Slamtec. It was responsible for generating the data used to create a digital map of the surroundings. It was an allothetic source of information with high accuracy and high speed of work due to the use of a laser beam in the infrared range as a data carrier. Table 4. presents the basic operating parameters of the RPLidar A1.

Table 4

BASIC OPERATING PARAMETERS FOR RPLIDAR A1

Parameter	Indoor operating mode	Field operation mode
Range	White objects: 25 m Black objects: 10 m	White objects: 25 m Black objects: 8 m
Scan frequency	Nominal value: 15 Hz, changeable in range: 5-20 Hz	
Sampling frequency	16 kHz	16 / 10 kHz
Angular resolution	0.225 °	0.225 / 0.36 °
Communication interface	TTL UART	
Transmission speed	256 000 bps	

Below is the fully assembled robot.

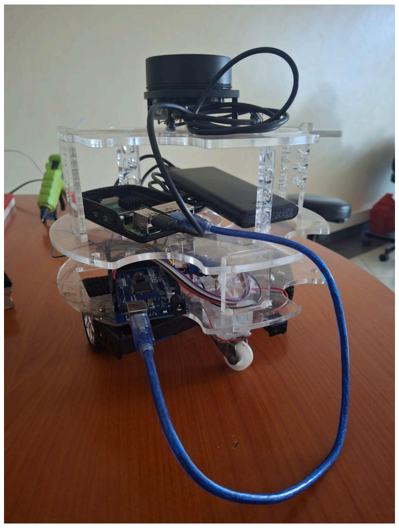

Fig. 4. Fully assembled Autonomous Robot

- Control and Software Components

The software control architecture was built around ROS2 (Robot Operating System 2), running on an Ubuntu OS installed on the Raspberry Pi 4. This decision was driven by ROS2's ability to handle real-time data processing and its modularity, which allowed us to integrate various sensors and control systems efficiently. The core control strategy utilized ROS2's SLAM and Nav2 (Navigation 2) packages to generate real-time maps of the gamefield and plan the robot's path through it. Rviz2 was used for data visualization, helping us track the robot's environment perception and verify its navigation in real-time.

1. *Ubuntu 22.04*

The Ubuntu system (Figure 2.5) is a complete GNU/Linux operating system distribution using MATE as the desktop environment.

Fig. 3. Screen of a computer with Ubuntu installed

This system provided great support for ROS2 software. This version of Ubuntu was the recommended one that was compatible with the specific ROS2 packages we desired to make use of. The same system was installed on the robot's platform. It was the latest version of this operating system, with the longest technical support at that moment. The TCP/IP network was used for communication between the robot's central unit and the user's personal computer. The communication was provided using the ssh communication protocol.

2. *Robot Operating System*

As previously mentioned, the software used to implement control algorithms and responsible for acquiring and processing of data from sensory systems was the ROS2 platform.

ROS2, the successor to the original ROS platform, was designed to overcome the limitations of ROS1 while enhancing its capabilities for more modern robotic systems. Released in 2017, ROS2 was created with the goal of improving performance, reliability, and scalability, especially in real-time and multi-robot systems. While

ROS1 was heavily dependent on a single communication layer, ROS2 adopted a Data Distribution Service (DDS) protocol to facilitate more robust and flexible communication between nodes, making it suitable for real-time applications and distributed systems.

Like ROS1, ROS2 operates using a modular architecture where various processes, known as nodes, run concurrently to manage different robot functionalities such as sensing, control, and navigation. These nodes are connected within a distributed system, allowing for seamless communication through topics, services, and actions. However, ROS2 improved upon ROS1 by adding support for native multi-threading, improved security features, and enhanced real-time performance.

Moreover, ROS2 is built with compatibility for more operating systems, including Windows, in addition to Ubuntu and macOS. It also includes enhanced support for embedded systems, making it a versatile choice for a wider range of robots, from mobile platforms to industrial and collaborative robots. ROS2 is a key enabler for modern robotic systems that require high reliability, real-time decision-making, and scalability across a variety of environments and applications.

The operation of the ROS platform was based on the communication of processes, which were simultaneously working nodes responsible for various functions of the robot (Fig. 2.6). They formed a network in which the processes were interconnected. This ensured that each node had access to the network, cooperation between them and the ability to monitor the type of data sent to the network. ROS enabled the exchange of messages in the form of topics, services, parameters and actions.

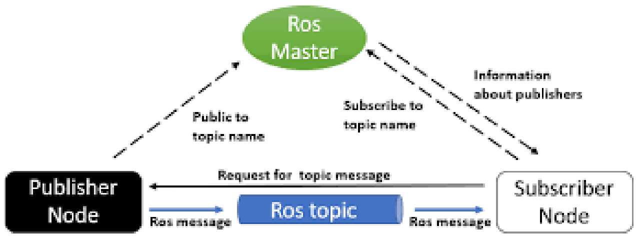

Fig. 5. Diagram of node cooperation in the ROS environment

In the ROS2 environment, programs are still developed using high-level languages like Python and C++, but the build system underwent significant changes. The primary build tool in ROS2 is colcon, which replaced catkin as the official build system. Colcon was developed to handle the increased complexity and scalability of ROS2, particularly when managing large workspaces with numerous packages.

Colcon, like catkin, automates the build process but introduces improved features for handling workspaces with multiple interdependent packages. It uses Python-based scripts and CMake macros to generate build rules tailored to the target environment, ensuring compatibility across different platforms. With colcon, users can build, test, and package multiple projects concurrently, making it highly efficient for large-scale projects. Additionally, colcon offers better dependency management, workspace overlaying, and improved support for cross-compilation.

Another advantage of colcon over its predecessor is its more flexible plugin-based architecture, allowing developers to extend its functionality easily. It also integrates well with modern development environments, streamlining the process of managing complex robotic systems in ROS2.

The ROS2 environment also had many graphical tools that allowed control of the operation of individual vehicle components and software packages. In addition, it was equipped with tools that allowed one to visualize vehicle operation on the basis of data from sensory systems, as well as to simulate the vehicle operation. The most commonly used tools were:

- ➔ RQT package – a platform included in the ROS software that enabled the implementation of various GUI tools in the form of plug-ins. The package was used to manage all tools in a single window. The platform was the most commonly used software for the diagnostics of robots. In addition, the platform allowed visualization of the structure of the nodes and the connections between them, which greatly facilitated the understanding of ROS operation. What is more, the RQT package enabled node analysis by mirroring the transformation tree launched by the basic node of the rqt_tf_tree system and invoking the node connection network – rqt_graph.
- ➔ Rviz2 (Fig. 6) – a package used for three-dimensional visualization of messages in ROS2, which was developed at a Korean university.

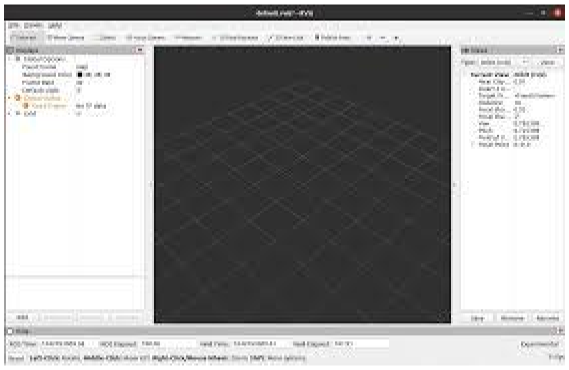

Fig. 7. Rviz graphic visualizer panel

It allows the visualization of data from sensors and depicts the robot's environment. In addition, it permits visualization of data from the perspective of a selected coordinate system based on data from the tf library. It also enabled graphical representation of the robot's URDF model and kinematic analysis of its motion. Additionally, the interface of the graphical tool enabled the creation of a digital map of the environment surrounding the robot based on sensor data.

➔ URDF (Unified Robot Description Format) – used to determine the kinematics and dynamics of the robot and was also used to represent the robot [23]. The format was used in the rviz visualizer and in the Gazebo simulator. URDF files were created using HTML. Based on the data contained in the unified format, it was possible to calculate the spatial pose of the robot and detect potential programming errors. URDF files with the appropriate plug-ins could be generated from CAD software.

An example description of a humanoid robot using URDF is shown in Fig. 8.

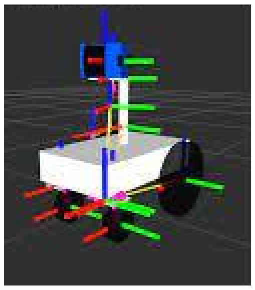

Fig. 8. Description of the robot representation using the URDF format

➔ Gazebo – a free ROS program for conducting 3D simulations of robot functioning, created in 2000 at the University of Southern California as a part of the Player project. It is one of the most popular robotics simulators that uses the OGRE (Object-Oriented Graphics Rendering Engine) graphics engine. This is because of its high efficiency and fully accurate mapping of reality and the laws of physics for which the ODE (Open Dynamics Engine) is responsible.

- *Algorithms*

A robot requires a finite sequence of predefined and consecutive actions in order to work correctly and complete its tasks. A simplified block diagram of the robot's algorithm that generates a 2D map, autonomous driving and remote control is shown in Fig. 7.

1. Control Algorithms

In ROS2, control algorithms for the robot include both autonomous navigation and remote control capabilities. Remote control was implemented via the ROS2 communication framework and the Secure Shell (SSH) protocol. Unlike ROS1, which relied on a central ROS Master for node communication, ROS2 adopts a decentralized model where nodes communicate directly with each other using DDS (Data Distribution Service) for discovery and message passing. This means there is no need for a single central node, and the system becomes more robust and fault-tolerant.

For remote control, the operator's computer interacts with the robot through this decentralized ROS2 network. Data exchange between the operator’s computer and the robot happens through ROS2 nodes in a peer-to-peer fashion, allowing real-time command transmission and feedback from the robot. Visualization and control of the robot’s operations are managed from the operator's unit using tools like Rviz2 or teleop, and the transmission remains duplex, enabling both control commands and sensor feedback.

In this setup, the network of nodes responsible for controlling the robot's actuators and sensors are interconnected, allowing the operator to monitor and adjust the robot's movements remotely. The use of ROS2's enhanced middleware also improves performance, allowing smoother remote operations, particularly in scenarios requiring low-latency control.

The network of nodes responsible for remote control of the effectors is presented in Fig. 8.

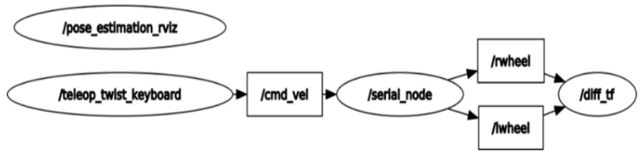

Fig. 9. Network of nodes responsible for remote control

In ROS2, the teleop_twist_keyboard node was responsible for handling user input from a personal computer, converting keystrokes into corresponding velocity commands in the form of geometry_msgs/Twist messages, which are published to the cmd_vel topic. These messages represent the robot's desired linear and angular velocities.

The cmd_vel messages are sent to the robot's onboard processing system, where a corresponding node is responsible for handling the serial communication with microcontrollers like the Arduino MEGA. Instead of the rosserial package used in ROS1, ROS2 utilizes micro-ROS or custom serial communication protocols to bridge the gap between ROS2 nodes and low-power devices like the Arduino. This allows the ROS2 nodes to convert high-level control commands into serial data, which is then sent via USB or other interfaces to the Arduino.

The diff_drive_controller in ROS2's ros2_control framework handles differential drive control. This node processes the cmd_vel messages and translates them into motor control signals, adjusting each wheel's speed accordingly. It also reads feedback from wheel encoders, providing odometry data, which is used for localization. This data is broadcasted as tf (transform) messages, allowing the navigation system to track the robot's position and orientation in real-time.

The robot's autonomous navigation is managed by Nav2, ROS2's equivalent of the ROS1 navigation_stack. Nav2 is responsible for path planning, local costmap management, and control, using sensor data such as Lidar and odometry to guide the robot to its destination. It generates the necessary control signals to the motors, ensuring the robot reaches the operator-defined goal while avoiding obstacles. This is done using advanced algorithms such as DWA (Dynamic Window Approach) or Teb (Timed Elastic Band) for path execution.

In ROS2, the Nav2 package serves as the main node for navigation, functioning similarly to the move_base node in ROS1. Like move_base, Nav2 combines both a global and local planner, while supporting two costmaps: local and global. These costmaps are essential components for navigation, providing the robot with information about obstacles in its immediate vicinity (local) and the overall environment (global).

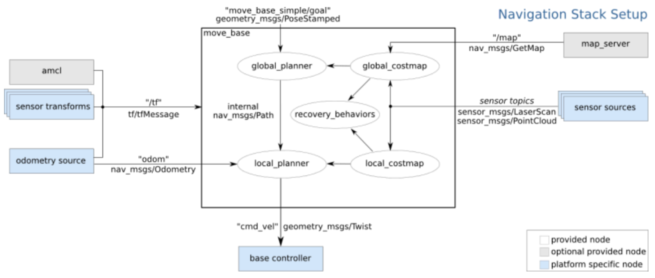

Fig. 10. Block diagram of the Nav2 stack.

The global planner in Nav2 is responsible for generating a path from the robot's current position to the target location. It typically uses algorithms like A* or Dijkstra's algorithm for this task, much like ROS1's move_base. Dijkstra's algorithm, a greedy algorithm, computes the shortest path between two points by evaluating all possible routes in a weighted graph, ensuring the path found has minimal cost.

The local planner in Nav2 uses the Dynamic Window Approach (DWA) for short-term trajectory generation. DWA, originally proposed by D. Fox, W. Burgard, and S. Thrun, computes control commands for the robot's linear and angular velocities in real-time. This algorithm samples the control space and simulates possible trajectories over short intervals, optimizing the robot's speed and proximity to the global path while avoiding obstacles. The steps in DWA remain the same in ROS2:

- Sampling possible velocities for the robot's control space.
- Simulating the effect of each sampled velocity on the robot's trajectory over a short time.
- Evaluating each trajectory based on its distance to the goal, proximity to obstacles, and speed.
- Selecting the best trajectory and converting it to control commands for the robot's motors.
- Executing the selected command and repeating the process in real-time.

In ROS2, Nav2 relies on several nodes, including the amcl node, which handles probabilistic localization using Adaptive Monte Carlo Localization(AMCL). This particle filter algorithm tracks the robot's position on a known map by sampling possible locations and weighing them based on sensor data, allowing the robot to estimate its pose even with noisy sensor readings. The localization process is integral to ensuring that the robot can navigate precisely in a 2D environment.

The robot's odometry is broadcasted through the tf library, which publishes coordinate transformations between the robot's various reference frames (e.g., base_link, wheels). The nav_msgs/Odometry message is used to publish this data, providing essential feedback for the navigation system. The differential drive controller reads this odometry information and ensures that the robot's movement aligns with the commanded velocities.

To avoid collisions, Nav2 integrates sensor data, such as from the Lidar sensor, which publishes sensor_msgs/LaserScan messages. These messages provide a 2D scan of the robot's surroundings, feeding obstacle data into the local costmap. This information allows the local planner (DWA) to react dynamically to obstacles by adjusting the robot’s velocity.

Control messages for the robot’s motors are sent using geometry_msgs/Twist messages, which contain fields for linear and angular velocities (e.g., `cmd_vel.linear.x`, `cmd_vel.linear.y`, and `cmd_vel.angular.z`). These messages are converted into motor commands by the low-level controllers, ensuring the robot moves as instructed by the navigation stack. They are sent in the form of a geometry_msgs/Twist message and converted into commands that properly control the operation of the motors. Figure 10. shows the network of operating nodes responsible for the correct implementation of autonomous navigation.

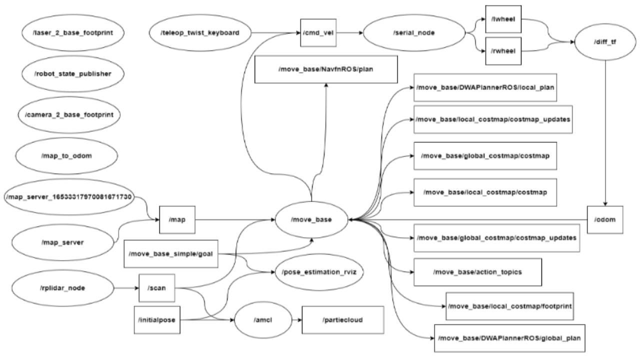

Fig. 11. Network of Operating Nodes for Autonomous Navigation

2. Map Generating Algorithms

In ROS2, the SLAM Toolbox has largely replaced gmapping as the go-to SLAM algorithm, though gmapping is still available for legacy projects. SLAM Toolbox is designed for both lifelong mapping and online SLAM, offering greater flexibility and performance improvements compared to gmapping. The algorithm can generate a two-dimensional map of the robot’s surroundings in real-time, using data from sensors such as Lidar.

While gmapping in ROS1 used an Extended Kalman Filter (EKF) to localize the robot, SLAM Toolbox in ROS2 employs more advanced techniques like Graph-based SLAM for better accuracy and scalability in complex environments. Compared to EKF-based methods, Graph SLAM is more robust when handling large and complex maps, avoiding the limitations of EKF that arise when linearizing the non-linear state estimation problem.

SLAM Toolbox also allows for easy map merging, saving, and continuous updates, making it more suitable for persistent mapping applications. Unlike hector_slam, which relies solely on Lidar data and does not use odometry, SLAM Toolbox can incorporate odometry, IMU (Inertial Measurement Unit), and laser scanner data, enhancing the quality and accuracy of the generated map. The use of SLAM Toolbox in ROS2 improves upon traditional SLAM methods like gmapping by providing higher accuracy, better resource management, and the ability to handle more dynamic environments, making it ideal for autonomous robot applications in both research and industrial settings.

The ROS2 software for generating a digital map, required an appropriate transformation of the coordinate systems representing each of the robot members. This was carried out by the tf library, which was responsible for the scene graph concept, reflected in the form of a hierarchy tree. The tree root was a representation of the environment map, and each of its vertices was a geometric transformation, translation, or rotation between the systems of each member . Figure 12 shows the transformation tree of the robot’s coordinate systems generated by the rqt_tf_tree package. The laser subtree reflected the coordinate system in a polar form represented by a laser scanner, while the base_link was a representation of the coordinate system of the robot platform. Odom was the frame responsible for locating the robot based on the data provided by the diff_tf node, which was responsible for acquiring data from encoders and transforming them into odometry information. The root of the tree was the map frame.

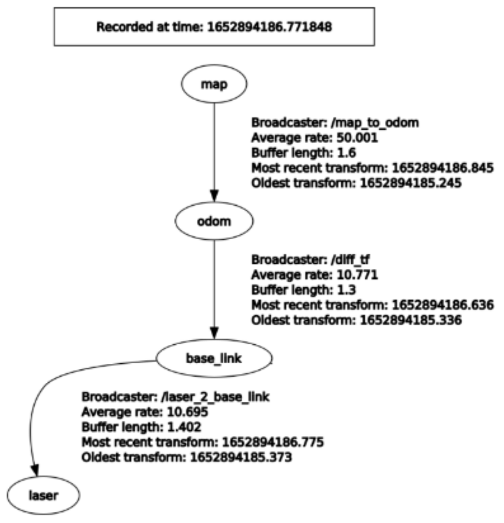
Recorded at time: 1652894186.771848
map
Broadcaster: /map_to_odom
Average rate: 50.001
Buffer length: 1.6
Most recent transform: 1652894186.845
Oldest transform: 1652894185.245
odom
Broadcaster: /diff_tf
Average rate: 10.771
Buffer length: 1.3
Most recent transform: 1652894186.636
Oldest transform: 1652894185.336
base_link
Broadcaster: /laser_2_base_link
Average rate: 10.695
Buffer length: 1.402
Most recent transform: 1652894186.775
Oldest transform: 1652894185.373
laser

Fig 12. Transformation Tree

C. *Experimental Results*

The team performed various tests to validate the functionality, reliability, and robustness of the autonomous robot. These tests include unit testing, integration testing, simulation testing, and preliminary reliability analysis. Below are the details and results of the conducted tests:

1. *Unit Testing*

Objective: Verify the correct operation of individual components (motors, sensors, communication modules).

- *Motor Testing*

Each encoded motor was tested independently by applying control signals via the L298N motor driver to ensure the correct speed and direction. The encoders were verified by monitoring feedback from the Arduino Mega, ensuring that the motor speed data matched the expected output.

Results: The motors were able to reach the desired speeds within a 2% margin of error. The encoder feedback correctly reported real-time speed data to ROS2, ensuring precise control.

- *Lidar (RPLidar A1) Testing*

The Lidar sensor was tested by capturing scan data in a controlled environment and analyzing the range and accuracy of detected objects.

Results: The Lidar was able to detect objects accurately up to its maximum range of 12 meters, with no significant deviations. However, minor data noise was observed when operating in highly reflective environments.

- *ROS2 Node Testing*

Each node responsible for motor control, sensor data processing, and communication was tested individually to ensure proper functionality and message flow.

Results: All nodes were confirmed to be operational, with messages successfully passed between publishers and subscribers. No significant message latency or packet loss was observed during the tests.

2. *Integration Testing*

Objective: Verify the interaction between hardware components and ROS2 nodes in real-world conditions.

- *Motor Control and Odometry Testing*

The motor control system, including the diff_drive_controller and odometry feedback, was tested in an open environment. The robot was tasked with moving in a straight line and performing rotational maneuvers based on velocity commands (`cmd_vel`).

Results: The robot achieved smooth forward motion and consistent rotational control, with deviations in movement under 3%. The odometry data matched the expected traveled distance, with a small drift observed after prolonged runs (about 1 cm drift after 5 meters).

- *Sensor Integration*

The Lidar and odometry data were integrated and tested using the SLAM algorithm. The robot was placed in a controlled environment with known obstacles, and its ability to map and localize itself was analyzed.

Results: The mapping process was successful, generating accurate 2D maps with a 95% match to the real environment. Minor localization errors were observed when the robot moved at higher speeds (>0.5 m/s), but this was mitigated by tuning the SLAM parameters.

- *Remote Control via Teleop Node*

The teleop_twist_keyboard node was used to control the robot remotely via an SSH connection, allowing manual commands to be sent over the ROS2 network.

Results: Remote control was responsive, with negligible latency (<100 ms delay) in command execution. The robot responded smoothly to velocity commands issued through the keyboard.

3. *Simulation Testing*

Objective: Evaluate the navigation system using simulated environments in Gazebo.

- *Navigation with Obstacles*

The robot was tested in a Gazebo simulation with various static and dynamic obstacles to assess its path-planning and collision avoidance abilities.

Results: The Nav2 stack performed well, with the global planner accurately planning paths around obstacles. The local planner (DWA) successfully avoided dynamic obstacles with only minor delays in reaction time (up to 0.2 seconds). In some cases, the robot had difficulty navigating narrow passages, but adjustments to the local costmap improved performance.

- *SLAM and Mapping*

The robot's mapping capabilities were tested by simulating different environments, including open areas and maze-like structures.

Results: SLAM was able to generate accurate maps in open areas, with minor discrepancies in tight or complex environments. The SLAM Toolbox demonstrated high performance with continuous mapping and loop closure, enabling map correction when revisiting previously mapped areas.

4. *Reliability and Robustness Testing*

Objective: Assess the reliability of the system over extended use and identify potential failure points.

- *Long-Run Motor and Odometry Test*

The robot was run continuously for 1 hour in a test field to measure wear on the motors and check odometry consistency over time.

Results: After 1 hour of continuous operation, motor performance remained stable. However, a slight degradation in odometry accuracy (2 cm drift per 10 meters) was noted, likely due to wheel slippage. The system remained operational with no hardware overheating.

- *Sensor Failure Simulation*

The team simulated a sensor failure scenario by temporarily disabling the Lidar sensor during operation, testing the system's behavior and recovery.

Results: The system recognized the loss of Lidar data and halted navigation as expected, triggering a fail-safe stop. Manual intervention was required to reset the sensor connection and resume operation.

- *Communication Fault Tolerance*

The robot's ability to handle communication interruptions between the Raspberry Pi and Arduino was tested by briefly severing the serial connection.

Results: The robot entered a safe mode upon losing communication, stopping all motor actions until the connection was restored. Upon re-establishment, the system resumed normal operation without manual intervention.

5. *Reliability and Robustness Studies*

The team conducted initial estimates of the robot's reliability by examining failure points, including potential sensor malfunctions and motor wear. Basic reliability modeling was performed using failure mode analysis, identifying the Lidar and motor drivers as critical components. Preliminary structural analysis of the acrylic chassis was also done to ensure durability under mechanical stress during the competition.

- *Failure Analysis*

The primary risk areas include Lidar sensor failure and motor driver overheating. Redundancy strategies, such as adding multiple sensors or introducing a backup communication protocol, are being explored.

- *Structural Analysis*

The acrylic chassis showed no signs of deformation or damage during testing, and the load-bearing capacity was deemed sufficient for the competition.

Summary of Testing Results:

- Motors: Accurate with minor drift over long distances.
- Lidar: Consistent, but slightly noisy in reflective environments.
- ROS2 Nodes: Reliable with minimal latency in communication.
- Navigation: Effective in both simulated and real environments, with improvements needed in narrow passage navigation.
- Reliability: High reliability with predictable failure modes and recovery mechanisms in place.

D. *Conclusion*

Robotics is increasingly influencing daily life, with engineers developing robots to take over labor-intensive, monotonous, and dangerous jobs. This trend is evident not only in industry but also in service sectors and private use. Alongside innovative designs, sophisticated software is essential for autonomous machines. An example of this advancement is the use of algorithms for SLAM (Simultaneous Localization and Mapping), enhancing robot autonomy and safety. The autonomous robot discussed in this article features advanced sensors, including the RPLidar A3M1 laser scanner, and employs control algorithms applicable to other mobile robots. Utilizing the LINUX OS and Robot Operating System, the robot operates in real-time, meeting requirements for autonomous navigation and two-dimensional map generation. Testing with the gmapping algorithm successfully enabled the robot to navigate between specified trajectory points. Additionally, thermal imaging assessments were conducted to evaluate cooling solutions for the onboard computer, comparing passive and active cooling efficiencies.

E. *References.*

[1] Slamtec. 2019. RPLIDAR 360 Degree Laser Range Scanner Interface Protocol and Application Notes. Shanghain Slamtec. Co.

[2] TECO ELECTRIC CO.: Motor Specification TFK280SC-21138-45. http://cdn.sparkfun.com/datasheets/Robotics/RP6%20motor%20TFK280SC-21138-45.pdf (access on 1.02.2022).

[3] L298N Motor Driver Controller Board. Instructables. 2015. https://www.makerfabs.com/l298n-motor-driver-board.html (access on 1.02.2022).

[4] Docter, Quentin, and Jon Buhagiar. 2019. Introduction to TCP / IP (from https://www.researchgate.net/publication/332460567_Introduction_to_TCPIP

[5] Quigley, Morgan, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. 2009. ROS: an open-source Robot Operating System. In Proceedings of the ICRA workshop on open source software 3 (3.2).

[6] Kam, Hyeong Ryeol, Sung Ho Lee, Taejung Park, and Chang Hun Ki. 2015. "RViz: a toolkit for real domain data visualization". Telecommunication Systems 60 (2) : 1-9.

[7] Kang, Yeon, Donghan Kim, and Kwangjin Kim. 2019. URDF Generator for Manipulated Robot. In Proceedings of the Third IEEE International Conference on Robotic Computing (IRC).

[8] Macenski, A. Soragna, M. Carroll, Z. Ge, "Impact of ROS 2 Node Composition in Robotic Systems", IEEE Robotics and Autonomous Letters (RA-L), 2023.

[9] K Belsare. Micro-ROS//A Koubaa. Robot Operating System (ROS). Cham:Springer International Publishing, 2023: 3-55.