Distributed Autonomous Exploration and Mapping: A ROS 2 Jazzy Approach with Thymio II

1st Marta Cardoso Lagarto Ferreira Valentim
Department of Informatics
Faculty of Sciences, University of Lisbon
Lisbon, Portugal
fc66100@alunos.fc.ul.pt

2nd Rafael Gomes Matias Ventura
Department of Informatics
Faculty of Sciences, University of Lisbon
Lisbon, Portugal
fc66132@alunos.fc.ul.pt

***Abstract*—This project presents the implementation of an autonomous mapping system built around the Thymio II educational robot and a Raspberry Pi 4, coordinated through the Robot Operating System (ROS 2 Jazzy). It tackles the challenge of extending a rather limited educational platform to perform SLAM (Simultaneous Localisation and Mapping) tasks. The system follows a distributed architecture, where high-bandwidth sensor processing — namely the LiDAR — runs containerised in Docker directly on the robot, while navigation logic and motor control are handled remotely from a central workstation. The report also explores in detail the ROS 2 concepts applied throughout development, particularly the use of Topics and TF (Transforms), and discusses how the slam_toolbox package was integrated into the overall workflow.**

***Index Terms*—ROS 2 Jazzy, Docker, Distributed Systems, SLAM, Thymio, LIDAR, Autonomous Exploration.**

I. Introduction

Mobile robotics operating in unstructured environments demands robust and reliable solutions for localisation, mapping, and path planning. The SLAM (Simultaneous Localisation and Mapping) problem raises a fundamental question: can a mobile robot, placed somewhere in an unknown environment, gradually build a consistent map while figuring out its own position within it? Essentially, the robot needs to tackle two closely related yet competing challenges — "Where am I?" (localisation) and "What does the world look like?" (mapping).

This project aims to bridge the gap between educational-grade hardware and more industrial-level software stacks to address this challenge in a practical way. By integrating a 360° LD06 LiDAR sensor with a Raspberry Pi 4, we equipped the educational Thymio robot with basic SLAM capabilities using the slam_toolbox package [1]. This work was developed as part of the Mobile Robotics course at the Faculty of Sciences, University of Lisbon.

II. System Architecture

A. Distributed Hardware Setup

The physical system is divided into two computational nodes to handle the specific requirements of the sensors:

1) **Perception Node (Raspberry Pi 4):** Mounted on the robot, this unit is dedicated to interfacing with the LD06 LIDAR. The sensor utilizes Time-of-Flight (ToF) technology to provide ranging data up to 12 meters [2].
2) **Control Node (Laptop):** Connects to the Thymio via its proprietary 2.4 GHz wireless dongle. It runs the heavy SLAM computations and the exploration logic.

To ensure autonomy, the Raspberry Pi is powered by an independent 5000mAh power bank, decoupling the computation power budget from the robot's motors.

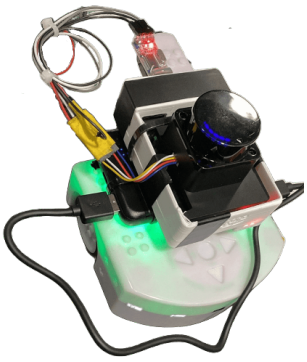

Fig. 1: The assembled prototype: Thymio II base, Raspberry Pi 4 in a custom 3D-printed case, and the LD06 LIDAR sensor. The green LEDs indicate active ROS 2 communication.

B. Data Flow Architecture

To visualize the distributed nature of the ROS 2 network, we designed a node graph architecture. This illustrates how the Raspberry Pi (Perception) and the Laptop (Control/SLAM) interact over DDS.

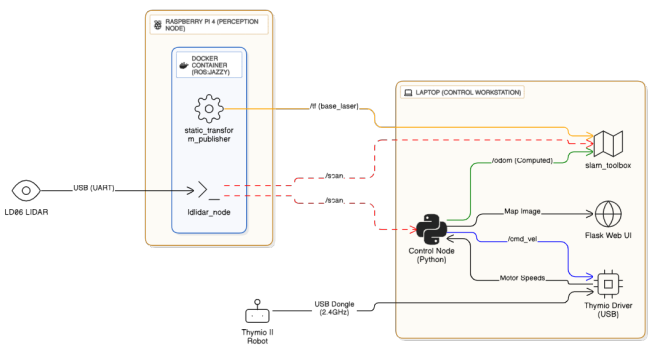

Fig. 2: System Node Graph designed in Eraser.io. The Raspberry Pi publishes LaserScans via Wi-Fi, while the Laptop handles Odometry, SLAM, and Command Velocity.

C. Network Configuration (ROS 2 & DDS)

The software framework is underpinned by fundamental ROS 2 concepts [3]. The development was managed within a standard workspace structure [4], where the code is organized into modular packages [5].

To enable communication between the Raspberry Pi and the Laptop over Wi-Fi, we utilized the ROS 2 middleware based on DDS (Data Distribution Service). As described in the ROS 2 configuration tutorials [6], ROS 2 provides a decentralized architecture ideal for distributed systems. A critical configuration was the assignment of a unique Domain ID:

```
export ROS_DOMAIN_ID=42          (1)
```

This environment variable was set on both machines, ensuring they discovered each other's topics without manual IP configuration.

III. Software Implementation

A. ROS 2 Core Concepts

The software architecture relies on two fundamental ROS 2 concepts:

1) Topics: Communication is achieved via Topics, acting as a named bus for message exchange [7]. In our system:

- `/scan`: Carries `LaserScan` data from the Pi to the Laptop.
- `/odom`: Carries the calculated robot position.
- `/cmd_vel`: Carries velocity commands to the motors.

2) TF (Transforms): For SLAM to function, the system must understand the robot's geometry. We utilized the TF system [8] to define the relationship between the robot center (`base_link`) and the sensor (`base_laser`). Since we utilized a custom mount, we defined a static transform of $z = 0.12m$.

B. Containerization (Docker)

To ensure a reproducible environment on the Raspberry Pi, we created a Docker container based on `ros:jazzy-ros-base`, following the best practices for ROS 2 Dockerfiles [9]. This isolates the dependencies for the LIDAR driver and TF publisher.

Listing 1: Dockerfile used on the Raspberry Pi

```
FROM ros:jazzy-ros-base

ENV ROS_DOMAIN_ID=42

RUN apt-get update && apt-get install -y \
    python3-pip \
    python3-colcon-common-extensions \
    python3-flask \
    python3-serial \
    python3-matplotlib \
    udev \
    git \
    && rm -rf /var/lib/apt/lists/*

# Install specific Thymio and TF libraries
RUN pip3 install thymiodirect transforms3d
    --break-system-packages

RUN echo "source /opt/ros/jazzy/setup.bash"
    >> /root/.bashrc

WORKDIR /root/robot_ws
```

Crucially, the container is deployed using the `--net=host` networking mode. This configuration is mandatory for ROS 2 DDS discovery over Wi-Fi, allowing multicast traffic to bypass Docker's default network isolation and reach the Control Node on the laptop.

C. Sensor Launch

On the Raspberry Pi, a Python launch file coordinates the `ldlidar_stl_ros2` driver [10] and the static transform publisher.

Listing 2: ROS 2 Launch File for Sensor and TF

```
import os
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    # LIDAR Driver Node
    ldlidar_node = Node(
        package='ldlidar_stl_ros2',
        executable='ldlidar_stl_ros2_node',
        name='LD06',
        output='screen',
        parameters=[
            {'product_name': 'LDLiDAR_LD06'},
            {'topic_name': 'scan'},
            {'frame_id': 'base_laser'},
            {'port_name': '/dev/ttyUSB0'},
            {'port_baudrate': 230400},
            {'laser_scan_dir': True},
            {'enable_angle_crop_func': False}
        ]
    )

    # Static Transform: Base to Laser (z=0.12m)
    base_to_laser_tf = Node(
        package='tf2_ros',
        executable='static_transform_publisher',
        arguments=['0.0', '0.0', '0.12',
                   '0.0', '0.0', '0.0',
                   'base_link', 'base_laser']
    )

    return LaunchDescription([
        base_to_laser_tf,
        ldlidar_node
    ])
```

D. Control Node Implementation

The Control Node, running on the laptop, serves as the central brain of the robot. Written in Python using the

`rclpy` library, it performs three critical functions: odometry computation, sensor fusion for navigation, and user interface management via Flask.

1) Custom Odometry Computation:

2) Custom Odometry Computation: Unlike standard ROS robots that output hardware odometry automatically, the Thymio requires manual computation. We implemented a differential drive kinematics model using the 2nd-order Runge-Kutta method for integration. The node reads the motor speeds via the `thymiodirect` library.

Listing 3: Runge-Kutta Integration Loop

```
# Constants from Lab 06
SPEED_COEFF = 0.0003027
ROBOT_WIDTH = 0.0935

def update_loop(self):
    # Read raw motor speeds
    rl = self.th[self.id]["motor.left.speed"]
    rr = self.th[self.id]["motor.right.speed"]

    # Handle 16-bit signed integer overflow
    if rl > 32767: rl -= 65536
    if rr > 32767: rr -= 65536

    # Calculate velocities
    vl = rl * SPEED_COEFF
    vr = rr * SPEED_COEFF
    v_lin = (vr + vl) / 2.0
    v_ang = (vr - vl) / ROBOT_WIDTH

    # 2nd order Runge-Kutta Integration
    delta_th = v_ang * dt
    mid_theta = self.th_yaw + (delta_th / 2.0)

    self.x += v_lin * math.cos(mid_theta) * dt
    self.y += v_lin * math.sin(mid_theta) * dt
    self.th_yaw += delta_th
```

In that study, we determined the linear speed coefficient K_d to be approximately 0.03027 (cm/unit/s) with a coefficient of variation of 13%. For the ROS implementation (in meters), we utilized:

$$K_d = 0.0003027 \quad [m/unit/s] \tag{2}$$

And the wheel baseline $L = 0.0935m$. To improve accuracy over simple Euler integration, we utilized the **Runge-Kutta 2nd Order (RK2)** method, as validated in Lab 05. First, linear (v) and angular (ω) velocities are derived:

$$v = \frac{(v_r + v_l) \times K_d}{2}, \quad \omega = \frac{(v_r - v_l) \times K_d}{L} \tag{3}$$

The pose is then integrated using the midpoint orientation θ_{mid}:

$$\theta_{mid} = \theta_t + \frac{\omega \Delta t}{2} \tag{4}$$

$$x_{t+1} = x_t + v \cos(\theta_{mid}) \Delta t \tag{5}$$

$$y_{t+1} = y_t + v \sin(\theta_{mid}) \Delta t \tag{6}$$

E. Odometry Drift Analysis

A critical observation is the performance discrepancy between the validated model from Lab 05 and the current project environment.

In Lab 05, conducted on a surface with adequate friction, we achieved high accuracy. For a target rotation, the calculated angular displacement was very close to the measured value [11], validating the kinematic constants.

However, in the current workshop environment characterized by smooth tiled flooring, we observed significant odometry drift due to wheel slippage. While the software model (Runge-Kutta) remains mathematically correct, the mechanical lack of traction resulted in noticeable rotational errors during operation. This confirms that dead-reckoning alone is insufficient for this environment, necessitating the use of *slam_toolbox*'s Loop Closure to correct the map.

1) Exploration Algorithm: We implemented a subsumption architecture for autonomous exploration. The logic prioritizes safety (IR sensors) over navigation (LIDAR).

Listing 4: Hybrid Navigation Logic

```
if max_ir > IR_THRESHOLD:
    # Layer 1: Emergency IR Avoidance (Close
        range)
    if weight_left > weight_right:
        tgt_l, tgt_r = ROTATION_SPEED,
            -ROTATION_SPEED # Turn Right
    else:
        tgt_l, tgt_r = -ROTATION_SPEED,
            ROTATION_SPEED # Turn Left

elif min_dist < LIDAR_WARN_DIST:
    # Layer 2: LIDAR Anticipation (Long range)
    print(f"[NAV] AVOIDING: {min_dist:.2f}m")
    if left_cone < right_cone:
        tgt_l, tgt_r = -10, ROBOT_SPEED # Avoid
            to Right
    else:
        tgt_l, tgt_r = ROBOT_SPEED, -10 # Avoid
            to Left
else:
    # Layer 3: Cruise Mode
    tgt_l, tgt_r = ROBOT_SPEED, ROBOT_SPEED
```

F. SLAM Configuration

We utilized the `slam_toolbox` in `online_async` mode. The configuration parameters were adapted from the reference configuration provided by Josh Newans in the *Articubot One* project [12], [13]. This reference was crucial for tuning the loop closure parameters to handle the noise inherent in our low-cost setup.

Listing 5: SLAM Toolbox Configuration (mapper_params_online_async.yaml)

```
slam_toolbox:
  ros__parameters:
    use_sim_time: False

    # Frames and Topics
    odom_frame: odom
```

```
map_frame: map
base_frame: base_link
scan_topic: /scan
mode: mapping

# Processing and Update Rates
transform_publish_period: 0.1
map_update_interval: 5.0
resolution: 0.05
max_laser_range: 12.0

# Movement Thresholds (Noise filtering)
minimum_travel_distance: 0.1
minimum_travel_heading: 0.1

# Loop Closure (Drift correction)
do_loop_closing: true
loop_search_maximum_distance: 3.0
loop_match_minimum_chain_size: 10
loop_match_maximum_variance_coarse: 3.0
loop_match_minimum_response_coarse: 0.35
loop_match_minimum_response_fine: 0.45

# Scan Matcher
distance_variance_penalty: 0.5
angle_variance_penalty: 1.0
fine_search_angle_offset: 0.00349
coarse_search_angle_offset: 0.349
coarse_angle_resolution: 0.0349
```

G. Real-time Map Visualization

Implement a pipeline to convert ROS data for web display. The concepts regarding occupancy grid manipulation were derived from the standard ROS map server documentation [14].

H. System Integration and Concurrency

The software architecture requires two blocking processes to run simultaneously: the ROS 2 event loop (to handle sensor callbacks and TF broadcasting) and the Flask HTTP server (to serve the user interface). To achieve this within a single Python executable, we utilized the `threading` module. The ROS 2 node is executed in a background daemon thread using `rclpy.spin()`, ensuring that sensor data is processed at 10Hz without interrupting the web server's request handling loop running on the main thread.

1) Occupancy Grid to Image Conversion: The `/map` topic provides an `OccupancyGrid` message, which represents the map as a linear array of probabilities (-1 for unknown, 0 for free, 100 for occupied). In the `get_map_image` endpoint:

1) **Reshaping:** The 1D array is reshaped into a 2D NumPy array using the map's metadata ($width \times height$).
2) **Normalization:** We apply a thresholding operation where values of 0 are mapped to 255 (White/Free space) and 100 to 0 (Black/Obstacles).
3) **Coordinate Flip:** Since ROS map coordinates start at the bottom-left and standard image formats start at the top-left, a vertical flip (`np.flipud`) is applied to ensure correct orientation.
4) **Encoding:** The processed array is converted to a grayscale PNG image in memory using the `PIL` library and served via HTTP, minimizing latency by avoiding disk I/O.

2) Command and Control: Beyond visualization, the interface provides control capabilities. The Flask API endpoints (`/start`, `/stop`) directly modify the internal state of the Control Node via the `is_exploring` flag. This allows the operator to trigger the autonomous behaviors described in Section IV-E or perform a soft emergency stop remotely.

IV. Experimental Setup

The tests were conducted in a workshop. The environment characterized by:

- **Flooring:** Surface covered with a slightly rough flooring that provides friction but allows the robot to walk well.
- **Obstacles:** A mix of static workshop equipment and an improvised course constructed from wooden planks.
- **Lighting:** Artificial indoor lighting.

V. Results and Discussion

A. Latency and Distributed Architecture

The transition to a distributed architecture was dictated by hardware constraints. Initially, running the full stack on the Raspberry Pi resulted in USB bottlenecks and serial communication timeouts due to the high interrupt load of the LIDAR (230400 baud). By offloading the Thymio driver to the laptop, we stabilized the system. However, this introduced a dependency on network latency. We observed that using `Reliability: Best Effort` for the LIDAR topic over DDS was crucial to maintain a fluid map update rate in RViz2 [7].

B. Odometry Performance: Lab 05 vs. Project Reality

A critical observation of this work is the discrepancy between the theoretical kinematic model and real-world performance on different surfaces.

In our previous work, Lab 05 [11], we empirically calibrated the linear speed coefficient (K_d) and achieved a consistent trajectory tracking with a coefficient of variation of only 13% on a high-friction surface. However, in the workshop environment characterized by smooth tiled flooring, we observed significant **odometry drift**.

Without the correctional feedback from the SLAM algorithm, the raw odometry indicated that the robot was rotating faster than it physically was. This phenomenon, caused by wheel slippage (low friction), resulted in "ghosting" artifacts in the generated map, where walls appeared duplicated or skewed by several degrees during rotation.

C. Impact of SLAM Tuning

To compensate for the mechanical slippage described above, the tuning of the *slam_toolbox* proved essential. By relaxing the strictness of the odometry trust (via the `angle_variance_penalty` parameter) and enabling `do_loop_closing`, we forced the system to rely on scan matching. This effectively corrected the drift: as the robot revisited a known area (loop closure), the algorithm detected the feature mismatch and "snapped" the map back to a coherent geometry, overriding the erroneous encoder data.

VI. Future Work

To address the limitations identified in the experimental phase, specifically the odometry drift on smooth surfaces, future iterations of this project could implement the following:

1) **Visual SLAM:** Utilizing the Raspberry Pi Camera (currently unused) to implement Visual Odometry or V-SLAM, adding a secondary layer of localization redundancy.
2) **Nav2 Integration:** Moving from reactive exploration to map-based path planning using the ROS 2 Navigation Stack (Nav2) for global goal planning.

VII. Conclusion

We successfully implemented a distributed robotic system capable of autonomous exploration. The use of Docker on the Raspberry Pi proved excellent for managing the software environment. However, the project highlighted the critical importance of mechanical traction in robotics; without reliable odometry (compromised by the tiled floor), even advanced SLAM algorithms struggle to maintain a coherent global map.

References

[1] S. Macenski, "Slam Toolbox: SLAM for the dynamic world," 2024. [Online]. Available: https://github.com/SteveMacenski/slam_toolbox

[2] J. Newans, "Hardware: LiDAR," 2024, articulated Robotics Tutorial. [Online]. Available: https://articulatedrobotics.xyz/tutorials/mobile-robot/hardware/lidar

[3] ——, "ROS 2 overview," 2024, articulated Robotics Tutorial. [Online]. Available: https://articulatedrobotics.xyz/tutorials/ready-for-ros/ros-overview/

[4] Open Robotics, "Creating a workspace," 2024, rOS 2 Jazzy Documentation. [Online]. Available: https://docs.ros.org/en/jazzy/Tutorials/Beginner-Client-Libraries/Creating-A-Workspace/Creating-A-Workspace.html

[5] J. Newans, "ROS 2 packages," 2024, articulated Robotics Tutorial. [Online]. Available: https://articulatedrobotics.xyz/tutorials/ready-for-ros/packages/

[6] Open Robotics, "Configuring ROS 2 environment," 2020. [Online]. Available: https://docs.ros.org/en/eloquent/Tutorials/Configuring-ROS2-Environment.html

[7] ——, "Understanding ROS 2 topics," 2024, rOS 2 Jazzy Documentation. [Online]. Available: https://docs.ros.org/en/jazzy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html

[8] J. Newans, "TF (transforms)," 2024, accessed: 2024-12-09. [Online]. Available: https://articulatedrobotics.xyz/tutorials/ready-for-ros/tf/

[9] ——, "Crafting a Dockerfile," 2024, articulated Robotics Tutorial. [Online]. Available: https://articulatedrobotics.xyz/tutorials/docker/crafting-dockerfile

[10] LDRobot Sensor Team, "LD19/LD06 LiDAR ROS 2 driver," 2024. [Online]. Available: https://github.com/ldrobotSensorTeam/ldlidar_stl_ros2

[11] M. Valentim and R. Ventura, "Odometry with Thymio," Faculty of Sciences, University of Lisbon, Lab Report 05, 2025, mobile Robotics Course.

[12] J. Newans, "Articubot One configuration repository," 2024, gitHub Repository. [Online]. Available: https://github.com/joshnewans/articubot_one/tree/main/config

[13] ——, "SLAM with ROS 2 - Articubot One," 2023, video Tutorial. [Online]. Available: https://www.youtube.com/watch?v=ZaiA3hWaRzE

[14] ROS Wiki, "map_server," 2024. [Online]. Available: https://wiki.ros.org/map_server