University of Liège

INFO0948-2 - Intelligent Robotics

TurtleBot3 Burger Project

Bruce Andriamampianina [S2302253]

Simon Gardier [s192580]

Arthur Graillet [S182019]

Pierre Sacré
Sven Goffin

Academic year : 2024-2025

Table des matières

1 Introduction

The aim of this project is to transform a TurtleBot into a deliveryman capable of autonomously delivering pizzas in an unknown environment. To do this, we must first build a map of the environment, locate the robot on it, identify the customers and memorize their pose on the map. In a second hand, the robot must be able to visit all customers of the list in the order to minimize its total delivery time.

2 Project architecture

2.1 Architecture diagram

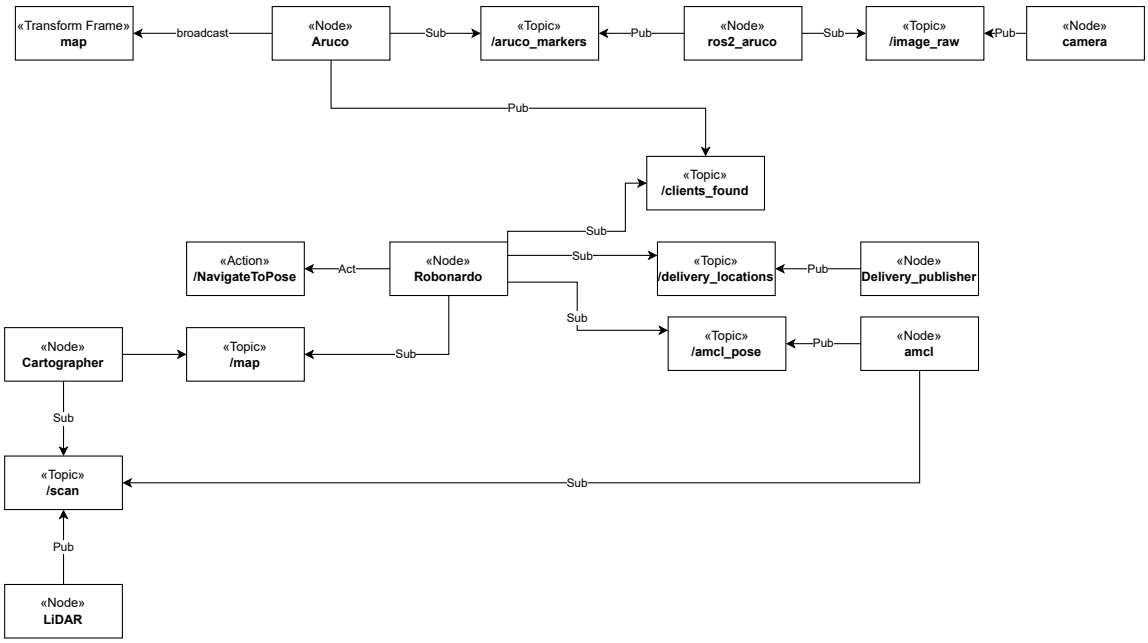

FIGURE 1 – Nodes and topics through which they communicate.

2.2 Launch order

Our project is launched in the following order

1. **Transform from camera to map frame** This is technically not a node but we start the project by launching `turtlebot3_state_publisher` which publishes the transform from `camera_link` to `map` in the TF2 tree.
2. **SLAM (Cartographer)**
   The Cartographer SLAM node builds a real-time map of the environment and publishes it on the `/map` topic. This topics is started using the turtlebot3 cartographer launch file, which also starts the next node : the AMCL node. It localizes the robot on the map and publish the estimated robot pose on `/amcl_pose`.

3. **Camera and marker detection (ros2_aruco)**
   The camera detects arucos and publishes their positions on Topics /aruco_markers
4. **Exploration and delivery (Robonardo)**
   Once mapping and localization has been launched, the main node takes care of the following tasks
   - Subscribes to `/map` and `/amcl_pose` to detect areas that not been detected by the LiDar and sends navigation goal through `/navigate_to_pose` actions.
   - Monitors `/clients_found` and `/delivery_locations` for delivery requests.
   - Pauses exploration when it receives a delivery request and the client has been found during mapping
   - Makes a complete turn on itself to detect all the arucos around it.
5. **Clients detection (Aruco)**
   This node retrieves the information on the arucos published on the Topics`/aruco_markers`, publishes static transforms `map → aruco_X` for each client, and updates the list on `/clients_found`.

3 Detection of clients

The `aruco` node detects clients using ArUco markers. When a marker is detected by the camera, the node transforms its position into the map frame and records it as a potential client. It also publishes the detected client IDs so that other system modules, such as delivery, can use this information.

3.1 Subscriptions and publications

The node subscribes to the `/aruco_markers` topic, which provides marker IDs and poses in the camera frame (`camera_optical_joint`). It publishes recognized client IDs on the `/clients_found` topic as an `Int32MultiArray`.

3.2 Clients pose detection and transform

For each detection, the node :

1. Creates a `PoseStamped` message with the marker pose in the camera frame.
2. Transform the pose from the `camera_link` frame to the map frame with **TF2** :

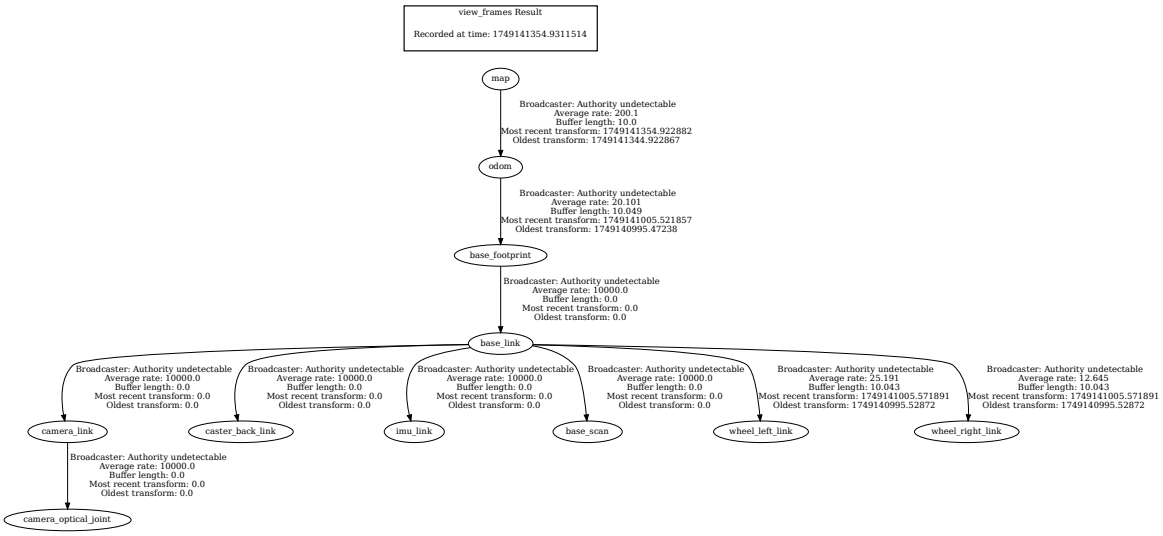

FIGURE 2 – TF2 tree of the robot

3. Averages the last 10 poses for the marker.
4. Broadcasts a static transform of this average from `map` to a child frame named `aruco_<id>_front` representing the delivery pose, 50cm in front of the client.
5. Publishes the new client IDs on `/clients_found`.

4 Autonomous environment exploration

Every 5 seconds, the robot schedule the exploration of the borders based on the last updates of the map.

4.1 Frontiers generation

A frontier cell is defined by the following criteria :

— The cell is free, with an occupancy equal to zero,
— It is adjacent to at least one unseen cell, which has an occupancy value of -1,
— It is sufficiently distant from obstacles, determined by checking occupancy values against a predefined safety threshold.

The occupancy grid is scanned cell-by-cell. Each candidate frontier cell's grid coordinates are converted into real-world metric coordinates, which are then stored in a frontier list for further processing.

4.2 Frontier goal selection

From the detected frontiers, the robot selects the target point closest to the robot's current position using A*.

4.3 Coordination with delivery

The exploration process can be temporarily paused by the delivery scheduling. This is implemented by setting an internal state flag that disables the exploration scheduling.

When a delivery task is received and the client pose is known, the exploration state is set to `paused`. During this pause, any current exploration navigation is canceled.

Once the delivery tasks are completed, the exploration is resumed, new frontier targets are generated and a new navigation action is issued.

The coordination is controlled by function calls to `pause_exploration_and_deliver_to()` and `resume_exploration()`).

5 Autonomous clients delivery

5.1 Receiving delivery requests

Delivery requests are received on the `/delivery_locations` topic, which provides client IDs awaiting service. These IDs are stored internally, when scheduling delivery the request are done according to the proximity of the client.

5.2 Planning and navigation

When a client in the delivery list is localized, the delivery scheduling is executed and if no other nearest delivery is in progress, Delivery to the customer can be initiated

During a delivery, exploration is put on hold, but this can be canceled if the robot receives an order from a customer closer to it than the customer it is currently looking after.

After delivering a customer, exploration is restarted if it has not been completed.

6 Issues and solutions

6.1 Incomplete detection of ArUco markers

— **Issue :** The robot finished exploring the occupancy grid but it failed to find all client, as some walls may not have been seen by the camera.
— **Solution :** Upon reaching any goal during exploration/delivery, the robot performs a 360-degree rotation to maximize visual coverage. The idea is that when robot reaches a frontier (= our goals exploration), doing a 360-degrees rotation should allow us to scan in a greedy fashion the environment.

Of course, even this solution is suboptimal : it does not ensure that all the arucos will be found. In practice we find between 6 to 9 arucos.

6.2 Collision with other turtlebot

— **Issue :** The LiDAR cannot detect objects below its scanning plane, such as the body of other robots. Only the top part (e.g., the turret) may be partially visible. Thus, the other robots are detect as really thin obstacle when in reality they are not.

— **Solution :** To account for this limitation, the robot's own radius in the costmap was doubled with an added safety margin. Indeed, this should avoid collision even if the other LiDAR is detected as a "single point".

6.3 Camera calibration

— **Issue :** During our tests, the camera calibration seemed odd as the estimated depth did not matched real measurements.

— **Solution :** Recalibration with the help of the `camera_calibration` node.

6.4 Imprecise ArUco pose estimation

— **Issue :** ArUco markers were sometimes loosely placed, causing imprecise delivery. The ArUco marker could be detected either in front of or behind the actual wall surface, leading to inaccurate position estimation.

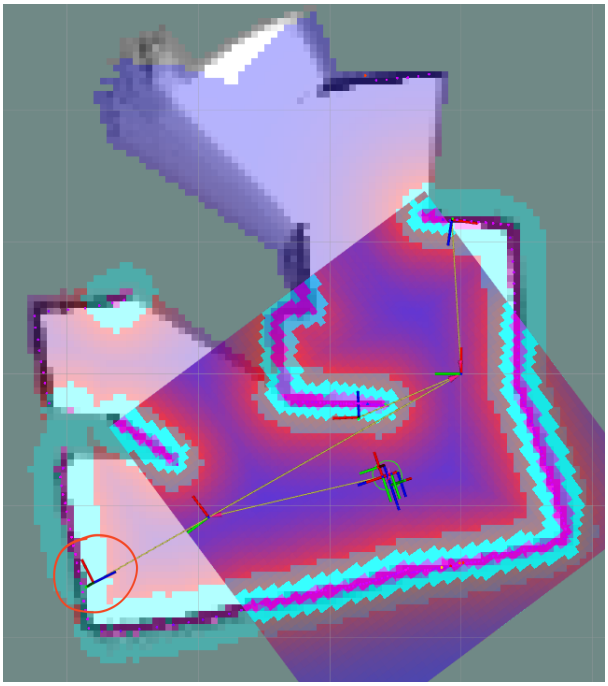

FIGURE 3 – Weak pose estimation in the bottom left

— **Solution :** The robot averages the last ten detected positions to smooth the pose estimate.

6.5 Exploration priority planning

— **Issue :** The system selected exploration goals based on Euclidean distance, sometimes based on choosing inaccessible path.

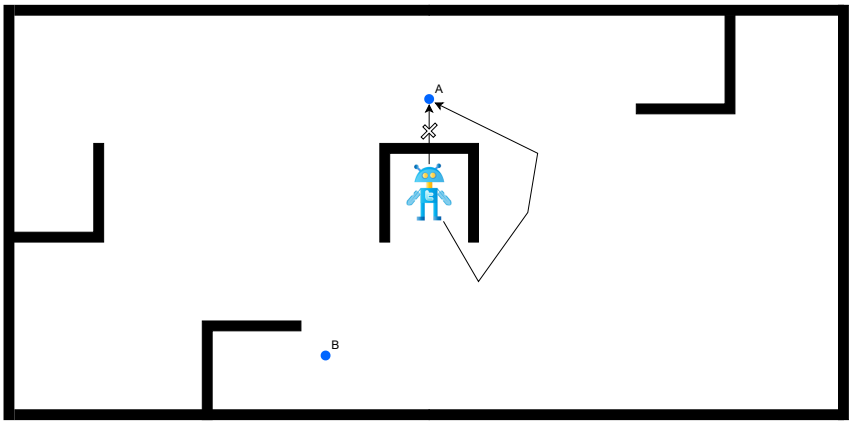

FIGURE 4 – Issue illustration

In this example, point A may appear to be the closest option based on Euclidean distance ; however, point B is clearly the more accessible choice when considering actual navigable paths.

— **Solution :** A new approach using A* has been implemented to prioritize reachable areas using real path planning cost instead of direct distance.

6.6 Dangerous path

— **Issue :** The path to the exploration goal or deliveries were often following the wall closely and the robot often ended colliding with those.

— **Solution :** The inflation radius and cost scaling factor parameters have been tuned to obtain a costmap with smooth transitions and no 0-cost regions. Changing from DWB to MPPI gave better trajectories and better obstacle avoidance. The MPPI algorithm has higher computation costs but the results are much better.

7 Results and improvements

7.1 Testing scenarios

The robot was tested in scenarios involving map exploration, client detection (ArUco markers), and dynamic delivery requests through the corresponding channel.

7.2 Observed results

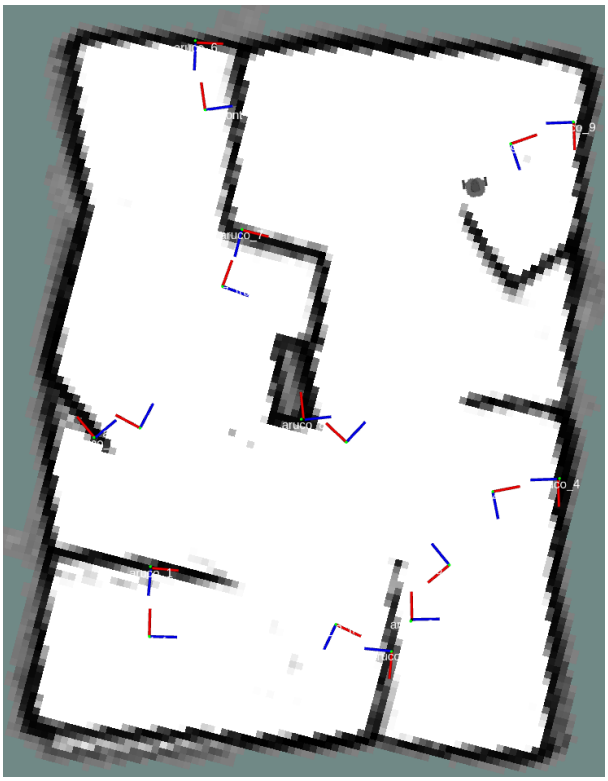

FIGURE 5 – Result of the exploration, the pose of the arucos detected and the pose for the delivery to these arucos.

— Exploration produced occupancy maps.
— ArUco detection became reliable after calibration and pose averaging.
— Delivery planning works according to knowledge of the clients and the robot location.
— The robot is capable of pausing exploration to deliver and then come back to exploration.

7.3 Further improvements

— **Obstacle perception :** Sometimes the robot does not register obstacles on its costmap because the obstacle is below the LiDAR or is too thin. Using computer vision to recognize obstacle.
— **Systematic wall scanning :** During exploration, the robot might fully map the occupancy grid without detecting all ArUco markers if their host walls were not properly scanned by the onboard camera. To solve this, one improvement would be to automatically place intermediate navigation goals facing newly discovered walls. Doing so would ensure that every wall would be seen at least once by the camera which would register any ArUco markers present on those.
— **Optimal delivery scheduling :** Our delivery scheduling algorithm always choose to deliver the nearest client. In practice this is often the best solution and is close to the optimal solution but one could creates scenarios where this approach would perform poorly.

A better solution would be to compute all the possible delivery orders and choose the one minimizing the distance across all the clients. We could also use the waiting time of the client in account as to prevent *starvation*. The minimization would be over the total waiting time of every client.

— **Reduce frontier selection compute cost :** Our exploration scheduling algorithm may generates over 100 frontiers for the robot to choose from. Then, the nearest frontier is selected by applying A* from the position to each frontier. The frontiers are often grouped together. One could implement a grouping algorithm to reduce the number of frontiers to the number of groups of frontiers. This would reduce the number of A* executions to select a frontier.

8 Conclusion

The final implementation is capable of exploring unknown environments, detecting ArUco markers, and navigate to the designated destinations.

Références

[1] Automatic Addison. ROS 2 Navigation Tuning Guide (Nav2). 2025. URL : https://automaticaddison.com/ros-2-navigation-tuning-guide-nav2/.

[2] Ani Arka. Custom ROS2 Explorer Script for Nav2. 2025. URL : https://github.com/AniArka/Autonomous-Explorer-and-Mapper-ros2-nav2/blob/main/custom_explorer/explorer.py.

[3] Navigation2 Contributors. Navigation2 Configuration Parameters. 2025. URL : https://github.com/ros-navigation/navigation2/blob/main/nav2_bringup/params/nav2_params.yaml.

[4] Navigation2 Documentation Team. Navigation2 Official Tuning Guide. 2025. URL : https://docs.nav2.org/tuning/index.html.

[5] Goffin Sven Pierre Sacré. Intelligent Robotics Guidelines. 2025. URL : https://gitlab.uliege.be/info0948-2/info0948-introduction-to-intelligent-robotics/-/tree/main/project.

[6] Open Robotics. ROS 2 Documentation Foxy. 2025. URL : https://docs.ros.org/en/foxy/Tutorials.html.

[7] Pierre Sacré et Sven Goffin. INFO0948 – Tutorial 2: Simulation with Gazebo. 2025. URL : https://gitlab.uliege.be/info0948-2/info0948-introduction-to-intelligent-robotics/-/tree/main/project.

[8] Pierre Sacré et Sven Goffin. INFO0948 – Tutorial 3: Turtlebot and SLAM. 2025. URL : https://gitlab.uliege.be/info0948-2/info0948-introduction-to-intelligent-robotics/-/blob/main/tutorials/tutorial-3.md.

[9] Pierre Sacré et Sven Goffin. INFO0948 – Tutorial 4: Turtlebot, Navigation, and ArUco Markers. 2025. URL : https://gitlab.uliege.be/info0948-2/info0948-introduction-to-intelligent-robotics/-/blob/main/tutorials/tutorial-4.md.