Robotics Track
Final Project- Navigation with NAV2 and Behavior Trees

Toni Tannous

Table of Contents

Research on NAV2, SLAM, AMCL, Path Planning, and BehaviorTree.CPP

Introduction

This research explores critical aspects of robotic navigation and decision-making, focusing primarily on the Navigation 2 (Nav2) stack, Simultaneous Localization and Mapping (SLAM), Adaptive Monte Carlo Localization (AMCL), path planning, and BehaviorTree.CPP. These components are essential for developing autonomous robotic systems capable of dynamic navigation and complex behavior management.

NAV2 Stack Overview

The Navigation 2 (Nav2) stack is an open-source ROS2-based project designed for robot navigation tasks, incorporating advanced algorithms for localization, mapping, and path planning. It allows robots to navigate environments autonomously, effectively avoiding obstacles and dynamically adjusting paths.

Key Components:

- **Localization:** Determines the robot's position within a known or unknown map.
- **Mapping:** Creates a representation of the environment for path planning.
- **Path Planning:** Calculates optimal paths from the robot's current position to a designated goal.
- **Control:** Implements real-time commands to navigate along the planned path.

SLAM (Simultaneous Localization and Mapping)

SLAM is a critical technique enabling robots to build a map of an unknown environment while simultaneously keeping track of their location within it.

SLAM Workflow:

1. **Data Collection:** Sensor data (e.g., LiDAR, camera, odometry) is collected.
2. **Map Creation:** The data is processed to build a coherent map.
3. **Localization:** Continuously updates the robot's estimated position relative to the created map.

Popular SLAM Methods:

- **Gmapping:** Grid-based SLAM using laser scans.
- **Cartographer:** Developed by Google, supports 2D and 3D SLAM.

Adaptive Monte Carlo Localization (AMCL)

AMCL is a probabilistic localization method based on particle filters. It estimates a robot's position within a known map using sensor data.

How AMCL Works:

- **Initialization:** Particles representing possible positions are distributed.
- **Prediction:** Particles are updated based on the robot's motion.
- **Update:** Particle weights are adjusted according to sensor data (e.g., LiDAR).
- **Resampling:** Particles are resampled based on their weights, concentrating on more probable locations.

Path Planning

Path planning determines the optimal route from an initial position to a goal, avoiding obstacles and minimizing costs such as distance or time.

Types of Path Planning Algorithms:

- **Global Planners:** (e.g., A*, Dijkstra) calculate an optimal path across the entire map.
- **Local Planners:** (e.g., DWB Planner, Pure Pursuit) dynamically adjust paths locally to handle unexpected obstacles.

BehaviorTree.CPP

BehaviorTree.CPP is a widely adopted C++ library for managing robot behavior using Behavior Trees within ROS2 environments. It structures robot decision-making processes clearly and efficiently, integrating seamlessly with the ROS2 navigation stack and visualization tools such as Groot.

Key Features of BehaviorTree.CPP:

- **Modularity:** Nodes can be easily created, reused, and composed to form complex behaviors.
- **Hierarchical structure:** Simplifies the organization and readability of complex decision-making tasks.
- **Flexibility:** Supports dynamic runtime modifications.
- **ROS2 Integration:** Includes built-in nodes for commonly used ROS2 navigation actions.

BehaviorTree.CPP Nodes:

BehaviorTree.CPP organizes behaviors using two primary node types:

- **Control Nodes:** Manage the execution flow, such as Sequence, Fallback, Parallel, and Decorator nodes.
- **Execution Nodes:** Perform actions or check conditions. Examples include Action and Condition nodes.

Integration with Nav2 Stack:

BehaviorTree.CPP integrates with Nav2 through asynchronous modular servers dedicated to control, planning, and recovery. Each node operates as a lifecycle-managed ROS2 node, allowing dynamic and runtime adjustments. It leverages multi-core processing capabilities and enhances the modularity and scalability of robotic navigation solutions.

Project Implementation

To implement this project, I first conducted research and then watched several videos on YouTube to better understand the key concepts

Creating the Map

The first task I completed was generating a map using ROS2 Nav2 with SLAM. To do this, I started by launching the simulation environment with the following command:

```
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

Next, I activated the SLAM functionality for the TurtleBot3 using this command:

```
ros2 launch turtlebot3_cartographer cartographer.launch.py use_sim_time:=True
```

Then, I moved the robot around the environment so that SLAM could scan and build the map. The SLAM algorithm detected obstacles (marked as black pixels), free spaces (white pixels), and unknown areas (gray pixels).

Once the map was fully built, I saved it using the following command:

```
ros2 run nav2_map_server map_saver_cli -f my_map
```

Note: The optional -f argument allows you to specify the path and name of the map file. There's no need to add an extension—this is done automatically.

Making the Robot Navigate Using the Map and ROS2 Nav2

To test navigation, I relaunched the simulation with:

```
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

In a new terminal, I started the Navigation2 stack and provided the saved map as an argument:

```
ros2 launch turtlebot3_navigation2 navigation2.launch.py
use_sim_time:=Truemap:=path/to/my_map.yaml
```

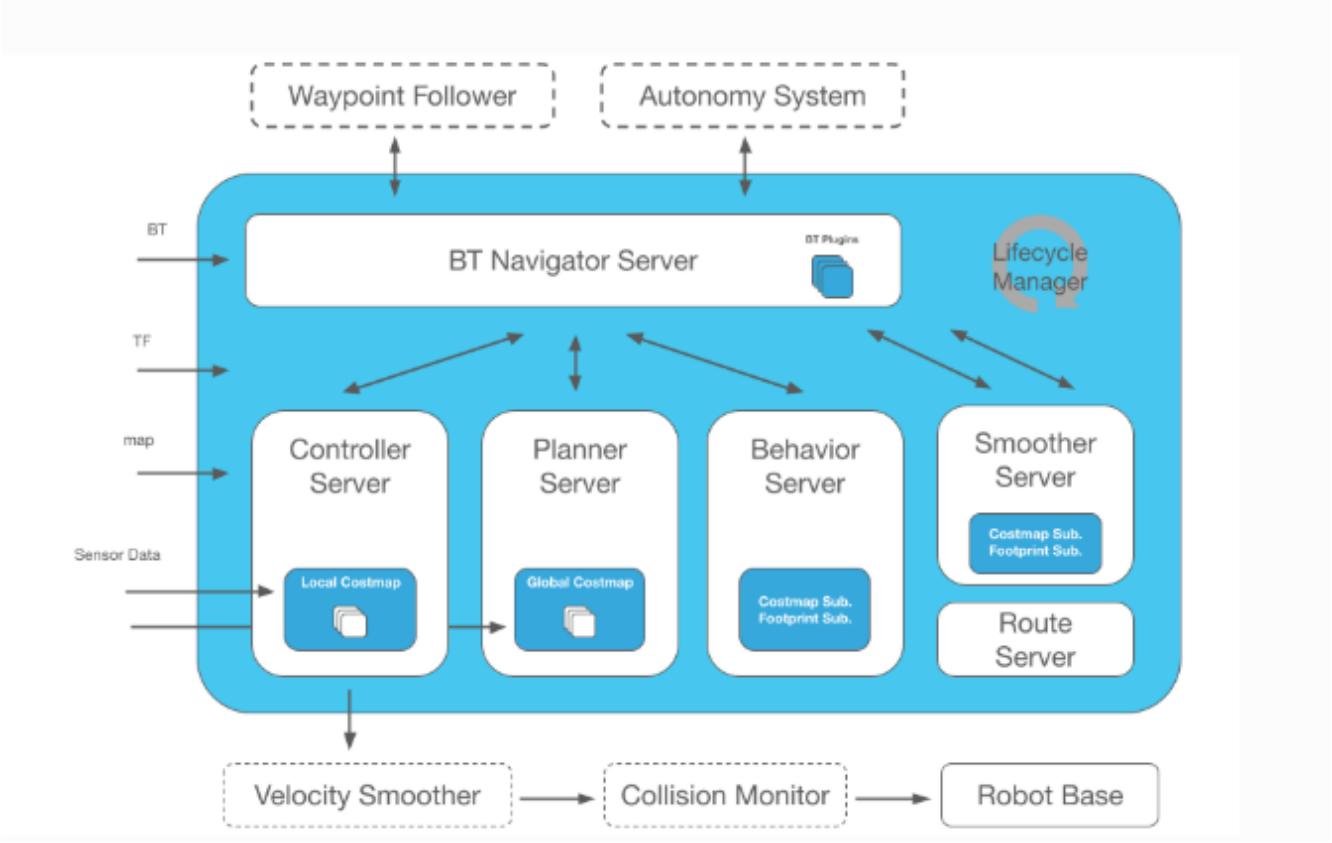
After that, I graphically localized the robot using the "2D Pose Estimate" tool and set navigation goals on the map interface. I observed that the robot successfully followed the goals, confirming that everything was working as expected.

Reflection

Although I primarily used the map for this project, this process helped me deepen my understanding of the concepts I had researched. It also gave me practical experience working with SLAM, map saving, and ROS2 navigation.

Communicating with the Nav2 Stack

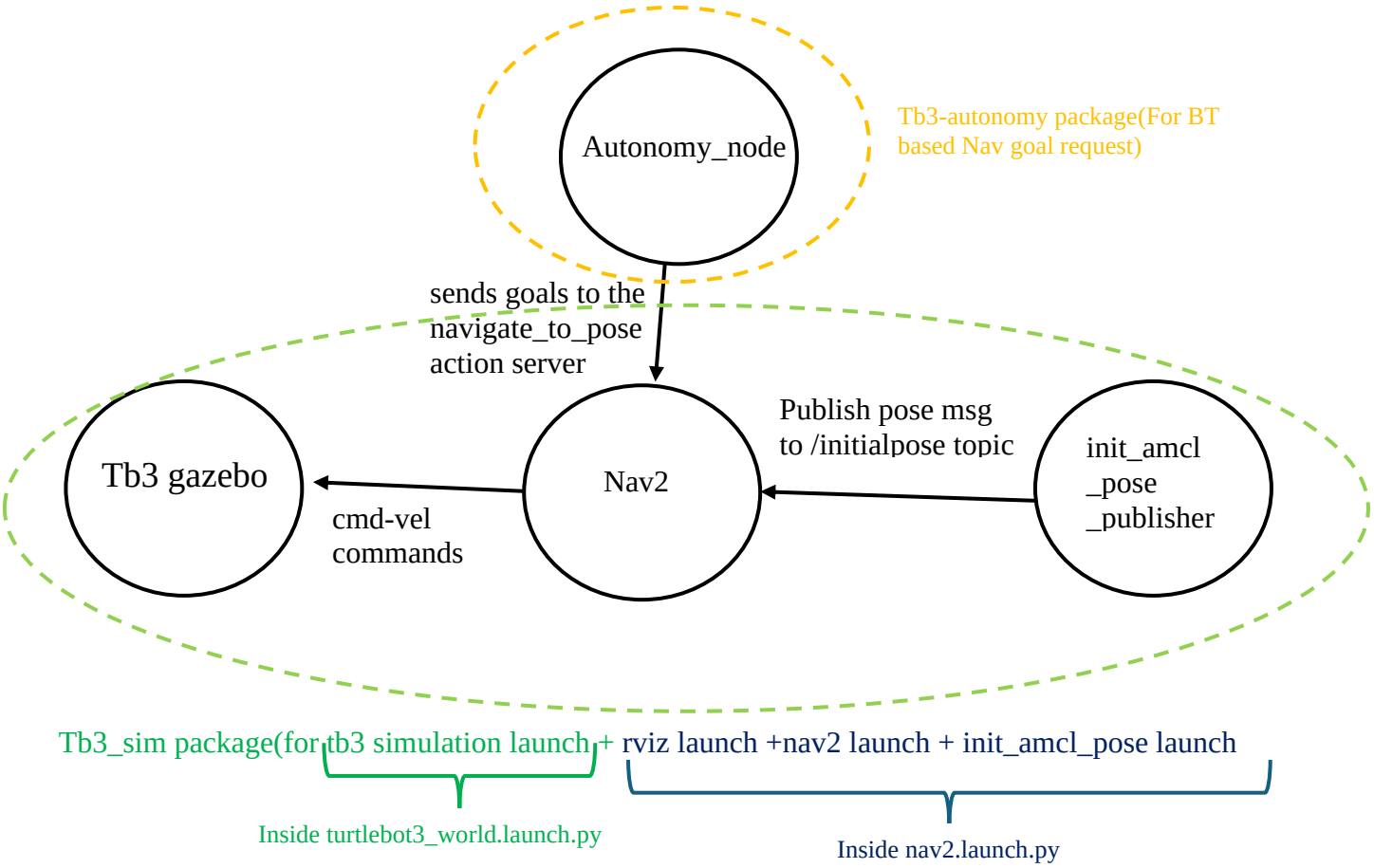
Now, I started working on the Project more seriously. I checked nav2 stack structure seen in the below image:



At the beginning of the project, I was using **RViz** to manually set the robot's initial pose and navigation goals. I used tools like **"2D Pose Estimate"** to set the starting position and **"Navigate To Pose"** to send goals. This helped me understand how the Nav2 system works on a graphical level.
As I progressed and wanted to make the robot navigate **automatically through code**, I started exploring how the internal components of the Nav2 stack communicate.
I discovered that the **BT Navigator Server** is responsible for receiving and handling navigation goals (waypoints). In my project, I created a **separate node** called **autonomy_node** that communicates with the BT Navigator Server.

To send waypoints to Nav2, I implemented an **action client** inside my autonomy_node. This client sends goals to the navigate_to_pose **action server**, which is part of the BT Navigator. With this setup, I no longer needed to rely on RViz—I could send goals programmatically.
In addition to sending waypoints, I also needed to set the robot's starting position. In RViz, this is usually done using the "2D Pose Estimate" tool, but I wanted to handle it through code. I found that the initial pose can be published to the /initialpose topic using the message type geometry_msgs/msg/PoseWithCovarianceStamped. So, I created a second node called **init_amcl_pose_publisher**, which automatically publishes the initial pose.
This transition from **graphical tools (RViz)** to **automated code-based control** allowed me to gain a much deeper understanding of how the Nav2 stack works behind the scenes.
Now, we will discuss the structure of the code.

Project Architecture Overview



The diagram above illustrates the overall architecture of the project and the interaction between its key components. The system is divided into two main packages: tb3_sim and tb3_autonomy. The tb3_sim package is responsible for launching the simulation environment, which includes Gazebo, RViz, Nav2,

and the init_amcl_pose_publisher node that publishes the robot's initial pose to the /initialpose topic. This setup is configured through turtlebot3_world.launch.py and nav2.launch.py.
On the other hand, the tb3_autonomy package contains the autonomy_node, which is responsible for sending navigation goals to the navigate_to_pose action server inside the Nav2 stack. Once Nav2 receives these goals, it processes them and sends velocity commands (cmd_vel) to control the robot in the Gazebo simulation.
This graph clearly demonstrates how each component contributes to the behavior-driven autonomous navigation of the TurtleBot3 robot.

Project Structure

The project is organized into multiple packages and directories to keep the navigation logic, simulation setup, and robot autonomy well-separated and modular. Below is a breakdown of the main components:

```
README.md
├── tb3_autonomy
│   ├── bt_xml
│   │   └── tree.xml
│   ├── CMakeLists.txt
│   ├── config
│   │   └── locations.yaml
│   ├── include
│   │   ├── autonomy_node.h
│   │   ├── battery_behavior.h
│   │   └── navigation_behaviors.h
│   ├── launch
│   │   └── autonomy.launch.py
│   ├── package.xml
│   ├── README.md
│   └── src
│       ├── autonomy_node.cpp
│       ├── battery_behavior.cpp
│       └── navigation_behaviors.cpp
├── tb3_sim
│   ├── CMakeLists.txt
│   ├── config
│   │   └── nav2_params.yaml
│   ├── launch
│   │   ├── nav2.launch.py
│   │   └── turtlebot3_world.launch.py
│   ├── maps
│   │   ├── map.pgm
│   │   └── map.yaml
│   ├── package.xml
│   ├── resource
│   │   └── tb3_sim
│   └── src
│       └── amcl_init_pose_publisher.cpp
├── turtlebot3
```

tb3_autonomy
This package contains the logic that defines the robot’s autonomous behaviors using a behavior tree and action clients.

- bt_xml/tree.xml: Defines the behavior tree structure.
- config/locations.yaml: Stores waypoint positions used in navigation.
- include/: Contains C++ header files for navigation, autonomy, and battery behaviors.
- src/: Includes source code for the main autonomy node (autonomy_node.cpp), navigation behavior (navigation_behaviors.cpp), and battery level management (battery_behavior.cpp).

- launch/autonomy.launch.py: Launches the full autonomy system with the behavior tree and required nodes.

tb3_sim

This package sets up the simulation environment for TurtleBot3.

- launch/nav2.launch.py: Starts the Nav2 stack + rviz + init_amcl_pose
- launch/turtlebot3_world.launch.py: Launches the simulated TurtleBot3 in a Gazebo world.
- config/nav2_params.yaml: Contains Nav2 parameters for costmaps, planners, and controllers.
- maps/: Stores the saved SLAM map (map.pgm and map.yaml).
- src/amcl_init_pose_publisher.cpp: Publishes the initial pose to /initialpose using AMCL.

turtlebot3

This is a dependency directory for the TurtleBot3 package files required to run the simulation. . I downloaded these packages using the turtlebot3.repos file, which can be found on the official TurtleBot3 GitHub repository. This file contains all the necessary repositories for the TurtleBot3 robot, including the simulation world, models, and navigation support. Using vcs made it easy to get everything I needed to run the environment properly.

```
vcs import . < turtlebot3.repos
```

This structure helps keep the simulation setup (tb3_sim) and the robot's autonomous logic (tb3_autonomy) cleanly separated. It also allows easy integration and testing of components like the behavior tree and waypoint navigation system.

Launching the Simulation and Testing the Setup

I started by creating the **tb3_sim** package, which is responsible for launching the simulation environment. To test it, I first launched the following file:

```
ros2 launch tb3_sim turtlebot3_world.launch.py
```

This command launches **Gazebo** with the TurtleBot3 model. Next, I launched:

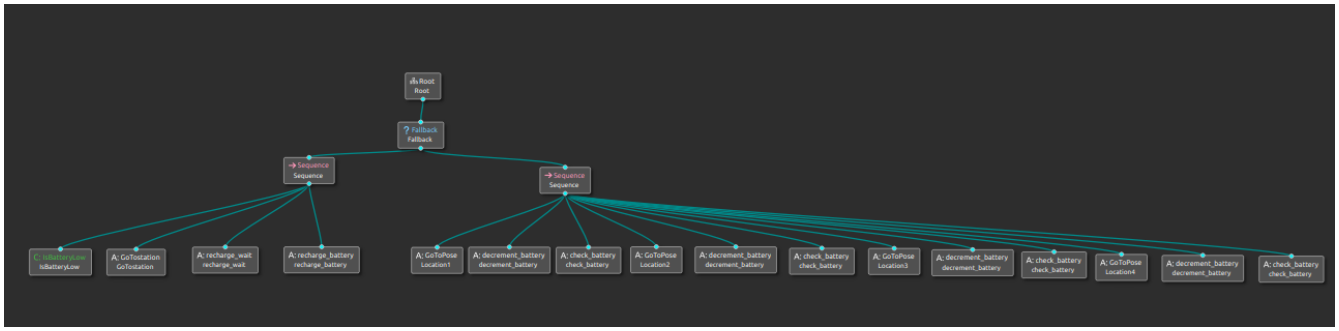
```
ros2 launch tb3_sim nav2.launch.py
```

This starts RViz, the Nav2 stack, and the amcl_init_pose_publisher node, which publishes the robot's initial pose to help localize it on the provided map (the same map I had previously created using SLAM).

After confirming that everything was working correctly by setting navigation goals through RViz and observing the robot successfully following the waypoints ,I moved on to building the second package, tb3_autonomy.

This package is where I implemented the robot’s autonomous behavior using a behavior tree. The tree defines the sequence and conditions under which the robot will move, recharge, and perform other tasks. I designed the behavior tree based on the desired behavior of the robot, and you can see the tree structure below.

Design and Logic of the Behavior Tree



The figure above shows the behavior tree I designed to control the robot's movement and battery-based decision-making. At the top level, the root node has a Fallback control node as its direct child. This node checks whether the robot should first **recharge** or proceed with **navigating to waypoints**.
The first child of the fallback is a **Sequence** node responsible for battery management. It begins by evaluating a **Condition** node: IsBatteryLow. If the battery is low, the robot navigates to a **charging station**, waits for 30 seconds to simulate recharging, and resets the battery level to 100%.
If the battery is **not** low, this sequence fails, triggering the **second child** of the fallback node: another **Sequence** node dedicated to waypoint navigation. In this sequence:

- The robot first moves to Location1 using the GoToPose action. This node uses a **custom action client** I implemented inside autonomy_node, which sends waypoints to Nav2's navigate_to_pose action server.
- When the goal is reached, the node returns success and activates the next node: decrement_battery, which subtracts 25 from the current battery level.
- After that, a condition node check_battery verifies if the battery level is zero.

Logically, the battery should never reach **exactly 0%** while navigating, because there should be enough charge left to **return to the station**. However, since this is a simulation, I simplified the logic and allowed the tree to detect a 0% battery and fail, triggering a return to the recharge sequence.
If the battery is **not** 0, the robot continues to the next location (e.g., Location2, Location3, etc.) and repeats the process. If the battery **is** 0, the condition fails, which causes the entire navigation sequence to fail. This failure bubbles up to the fallback and root nodes, causing the tree to restart(I have specified this in the code). On the next cycle, IsBatteryLow will now return true, and the robot will go to the station to recharge and repeat the cycle.

Accessing the Source Code

The code details will not be explained in this report. However, the entire implementation is clearly documented with comments directly in the source files. You can find the full project and all related files on the GitHub repository:
https://github.com/Tannous987/nav_tb3_ws.git

References

https://www.behaviortree.dev/

https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping

https://neobotix-docs.de/ros/ros2/autonomous_navigation.html#change-parameters-and-algorithms

https://docs.nav2.org/

https://roboticsbackend.com/ros2-nav2-tutorial/

https://docs.nav2.org/tutorials/docs/navigation2_with_slam.html

https://docs.nav2.org/configuration/packages/configuring-amcl.html