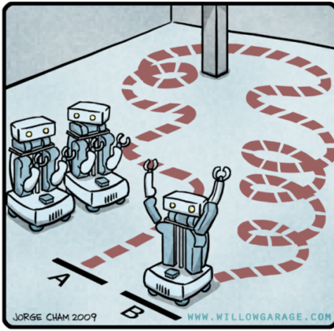
ROS 2 Navigation

Gianluca Palli

gianluca.palli@unibo.it

DEI - LAR
University of Bologna
Viale del Risorgimento 2
40136 Bologna

R.O.B.O.T. Comics



"HIS PATH-PLANNING MAY BE SUB-OPTIMAL, BUT IT'S GOT FLAIR."

Navigation Concepts

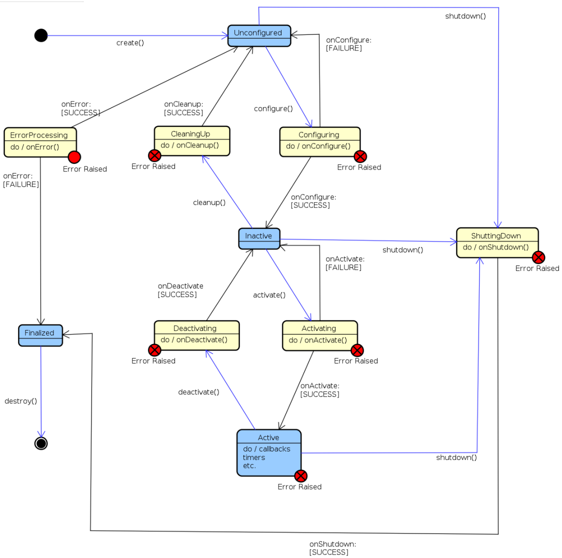
Action servers are used in the navigation stack to communicate with the highest level Behavior Tree (BT) navigator through a `NavigateToPose` action message

They are also used for the BT navigator to communicate with the subsequent smaller action servers to compute plans, control efforts, and recoveries. Each will have their own unique `.action` type in `nav2_msgs` for interacting with the servers

Lifecycle nodes contain state machine transitions for bringup and teardown of ROS 2 servers. This helps in determinstic behavior of ROS systems in startup and shutdown. It also helps users structure their programs in reasonable ways for commercial uses and debugging

Behavior trees are a tree structure of tasks to be completed. It creates a more scalable and human-understandable framework for defining multi-step or many state applications. This is opposed to a finite state machine (FSM) which may then have dozens or states and hundreds of transitions

Lifecycle Nodes



Navigation Servers - Planners

The task of a **planner** is to compute a path to complete some objective function. The path can also be known as a route, depending on the nomenclature and algorithm selected. Two canonical examples are computing a plan to a goal (e.g. from current position to a goal) or complete coverage (e.g. plan to cover all free space). The planner will have access to a global environmental representation and sensor data buffered into it. Planners can be written to:

- Compute shortest path
- Compute complete coverage path
- Compute paths along sparse or predefined routes

The general task in Nav2 for the planner is to compute a valid, and potentially optimal, path from the current pose to a goal pose. However, many classes of plans and routes exist which are supported.

Navigation Servers - Controllers

Controllers are the way we follow the globally computed path or complete a local task. The controller will have access to a local environment representation to attempt to compute feasible control efforts for the base to follow. Many controller will project the robot forward in space and compute a locally feasible path at each update iteration. Controllers can be written to:

- Follow a path
- Dock with a charging station using detectors in the odometric frame
- Board an elevator
- Interface with a tool

The general task in Nav2 for a controller is to compute a valid control effort to follow the global plan. However, many classes of controllers and local planners exist. It is the goal of this project that all controller algorithms can be plugins in this server for common research and industrial tasks.

Navigation Servers - Recoveries

Recoveries are a mainstay of fault-tolerant systems. The goal of recoveries are to deal with unknown or failure conditions of the system and autonomously handle them. Examples may include faults in the perception system resulting in the environmental representation being full of fake obstacles. The clear costmap recovery would then be triggered to allow the robot to move.

Another example would be if the robot was stuck due to dynamic obstacles or poor control. Backing up or spinning in place, if permissible, allow the robot to move from a poor location into free space it may navigate successfully.

Finally, in the case of a total failure, a recovery may be implemented to call an operators attention for help. This can be done with email, SMS, Slack, Matrix, etc.

Navigation Servers - Waypoint Following

The `nav2_waypoint_follower` contains a **waypoint following** program with a plugin interface for specific task executors. This is useful if you need to go to a given location and complete a specific task like take a picture, pick up a box, or wait for user input. It is a nice demo application for how to use Nav2 in a sample application.

However, it could be used for more than just a sample application. There are 2 schools of thoughts for fleet managers / dispatchers.

- Dumb robot; smart centralized dispatcher
- Smart robot; dumb centralized dispatcher

In the first, the `nav2_waypoint_follower` is weakly sufficient to create a production-grade on-robot solution. The application on the robot just needs to worry about the task at hand and not the other complexities of the system complete the requested task.

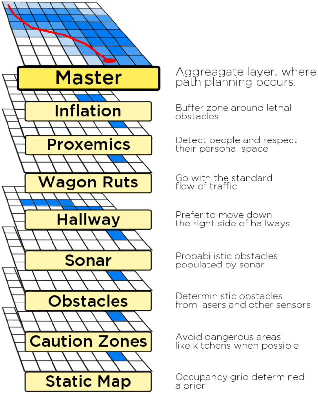
In the second, the `nav2_waypoint_follower` is a nice sample application / proof of concept, but you really need your waypoint following / autonomy system on the robot to carry more weight in making a robust solution.

Costmaps and Layers

The current environmental representation is a **costmap**

- A costmap is a regular 2D grid of cells containing a cost from unknown, free, occupied, or inflated cost
- This costmap is then searched to compute a global plan or sampled to compute local control efforts

Various costmap layers are implemented as pluginlib plugins to buffer information into the costmap. This includes information from LIDAR, RADAR, sonar, depth, images, etc. It may be wise to process sensor data before inputting it into the costmap layer, but that is up to the developer

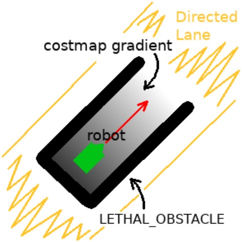


Costmaps Filters

Costmap filters - is costmap layer based approach of applying spatial-dependent behavioral changes annotated in filter masks, into Nav2 stack. Costmap filters are implemented as costmap plugins

For example, the following functionality could be made by using of costmap filters:

- Keep-out/safety zones where robots will never enter
- Speed restriction areas. Maximum speed of robots going inside those areas will be limited
- Preferred lanes for robots moving in industrial environments and warehouses



State Estimation

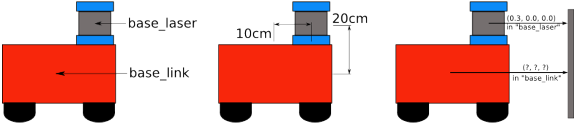
There are 2 major transformation frames that need to be provided, according to community standards

- `map` to `odom` transform is provided by a positioning system (localization, mapping, SLAM)
- `odom` to `base_link` by an odometry system

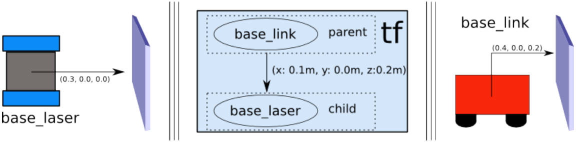
At minimum, a TF tree that contains a full `map -> odom -> base_link -> [sensor frames]` for your robot is needed

Setting Up Transformations

This robot has two defined coordinate frames: `base_link` corresponding to the center point of the mobile base of the robot, and `base_laser` for the center point of the laser that is mounted on top of the base



The transform associated with the edge connecting `base_link` and `base_laser` should be (x: 0.1m, y: 0.0m, z: 0.2m)



With this transform tree set up, converting the laser scan received in the `base_laser` frame to the `base_link` frame is as simple as making a call to the TF2 library

Static Transform Publisher Demo

Now let's try publishing a very simple transform using the `static_transform_publisher` tool provided by TF2

Open up your command line and execute the following command:

```
$ ros2 run tf2_ros static_transform_publisher \
    0.1 0 0.2 0 0 0 base_link base_laser
```

With this, we are now sucessfully publishing our `base_link` to `base_laser` transform in TF2. Let us now check if it is working properly through `tf2_echo`

Open up a separate command line window and execute the following:

```
$ ros2 run tf2_ros tf2_echo base_link base_laser
```

You should see

```
At time 0.0
- Translation: [0.100, 0.000, 0.200]
- Rotation: in Quaternion [0.000, 0.000, 0.000, 1.000]
```

Setting Up the URDF

Let's begin by installing some additional ROS 2 packages

```
$ sudo apt install ros-foxy-joint-state-publisher-gui
$ sudo apt install ros-foxy-xacro
```

Next, create a directory for your project, initialize a ROS 2 workspace and give your robot a name like `sam_bot`

```
$ ros2 pkg create --build-type ament_cmake sam_bot_description
```

To get started, create a file named `sam_bot_description.urdf` under `src/description` and input the following as the initial contents of the file

```xml
<?xml version="1.0"?>
<robot name="sam_bot" xmlns:xacro="http://ros.org/wiki/xacro">

</robot>
```

Setting Up the URDF (Cont.)

Next, let us define some constants using XAcro properties that will be reused throughout the URDF

```xml
<!-- Define robot constants -->
<xacro:property name="base_width" value="0.31"/>
<xacro:property name="base_length" value="0.42"/>
<xacro:property name="base_height" value="0.18"/>

<xacro:property name="wheel_radius" value="0.10"/>
<xacro:property name="wheel_width" value="0.04"/>
<xacro:property name="wheel_ygap" value="0.025"/>
<xacro:property name="wheel_zoff" value="0.05"/>
<xacro:property name="wheel_xoff" value="0.12"/>

<xacro:property name="caster_xoff" value="0.14"/>
```

Setting Up the URDF (Cont.)

Let us then define our `base_link` - this link will be a large box and will act as the main chassis of our robot

```
<!-- Robot Base -->
<link name="base_link">
  <visual>
    <geometry>
      <box size="${base_length} ${base_width} ${base_height}"/>
    </geometry>
    <material name="Cyan">
      <color rgba="0 1.0 1.0 1.0"/>
    </material>
  </visual>
</link>
```

Setting Up the URDF (Cont.)

Next, let us define a base_footprint, a virtual (non-physical) link which has no dimensions or collision areas representing the center of a robot projected to the ground

Nav2 uses this link to determine the center of a circular footprint used in its obstacle avoidance algorithms

```xml
<!-- Robot Footprint -->
<link name="base_footprint"/>

<joint name="base_joint" type="fixed">
  <parent link="base_footprint"/>
  <child link="base_link"/>
  <origin xyz="0.0 0.0 ${(wheel_radius+wheel_zoff)}" rpy="0 0 0"/>
</joint>
```

After defining our base_link, we then add a joint to connect it to base_link

Setting Up the URDF (Cont.)

Let's use a macro to define a generic wheel

```
<!-- Wheels -->
<xacro:macro name="wheel" params="prefix x_reflect y_reflect">
  <link name="${prefix}_link">
    <visual>
      <origin xyz="0 0 0" rpy="${pi/2} 0 0"/>
      <geometry>
          <cylinder radius="${wheel_radius}" length="${wheel_width}"/>
      </geometry>
      <material name="Gray">
        <color rgba="0.5 0.5 0.5 1.0"/>
      </material>
    </visual>
  </link>

  <joint name="${prefix}_joint" type="continuous">
    <parent link="base_link"/>
    <child link="${prefix}_link"/>
    <origin xyz="${x_reflect*wheel_xoff} \
   ${y_reflect*(base_width/2+wheel_ygap)} ${-wheel_zoff}" rpy="0 0 0"/>
    <axis xyz="0 1 0"/>
  </joint>
</xacro:macro>
```

Setting Up the URDF (Cont.)

Now, we will be adding two large drive wheels to our robot

```
<xacro:wheel prefix="drivewhl_l" x_reflect="-1" y_reflect="1" />
<xacro:wheel prefix="drivewhl_r" x_reflect="-1" y_reflect="-1" />
```

Next, we will be adding a caster wheel at the front of our robot

```
<!-- Caster Wheel -->
<link name="front_caster">
  <visual>
    <geometry>
      <sphere radius="${(wheel_radius+wheel_zoff-(base_height/2))}"/>
    </geometry>
    <material name="Cyan">
      <color rgba="0 1.0 1.0 1.0"/>
    </material>
  </visual>
</link>

<joint name="caster_joint" type="fixed">
  <parent link="base_link"/>
  <child link="front_caster"/>
  <origin xyz="${caster_xoff} 0.0 ${-(base_height/2)}" rpy="0 0 0"/>
</joint>
```

Package Dependancies

Open up the root of your project directory and add the following lines to your package.xml after the <buildtool_depend> tag

```
<exec_depend>joint_state_publisher</exec_depend>
<exec_depend>joint_state_publisher_gui</exec_depend>
<exec_depend>robot_state_publisher</exec_depend>
<exec_depend>rviz</exec_depend>
<exec_depend>xacro</exec_depend>
```

Let us modify the CMakeLists.txt file in the project root directory to include the files we just created during the package installation process. Add the following snippet above the if(BUILD_TESTING) line:

```
install(
  DIRECTORY src launch rviz description
  DESTINATION share/${PROJECT_NAME}
)
```

Launch File

From the root of the project, create a directory named `launch` and a `display.launch.py` file within it

```
import launch
from launch.substitutions import Command, LaunchConfiguration
import launch_ros
import os

def generate_launch_description():
  pkg_share = launch_ros.substitutions.FindPackageShare \
     (package='sam_bot_description').find('sam_bot_description')
  default_model_path = os.path.join(pkg_share, \
    'description/sam_bot_description.urdf')
  default_rviz_config_path = os.path.join(pkg_share, \
    'rviz/urdf_config.rviz')

  robot_state_publisher_node = launch_ros.actions.Node(
    package='robot_state_publisher',
    executable='robot_state_publisher',
    parameters=[{'robot_description': Command(['xacro ', \
      LaunchConfiguration('model')])}]
  )
  ...
```

Launch File (Cont.)

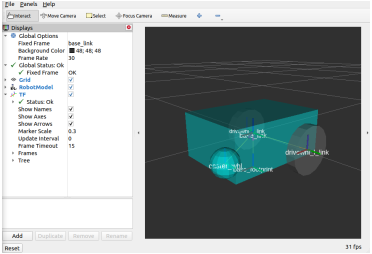
```
joint_state_publisher_node = launch_ros.actions.Node(
  package='joint_state_publisher',
  executable='joint_state_publisher',
  name='joint_state_publisher',
  condition=launch.conditions.UnlessCondition(LaunchConfiguration('gui'))
)
joint_state_publisher_gui_node = launch_ros.actions.Node(
  package='joint_state_publisher_gui',
  executable='joint_state_publisher_gui',
  name='joint_state_publisher_gui',
  condition=launch.conditions.IfCondition(LaunchConfiguration('gui'))
)
rviz_node = launch_ros.actions.Node(
  package='rviz2',
  executable='rviz2',
  name='rviz2',
  output='screen',
  arguments=['-d', LaunchConfiguration('rvizconfig')],
)

...
```

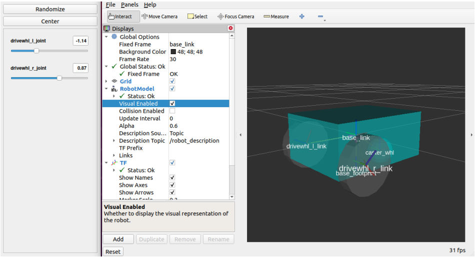
Launch File (Cont.)

```
return launch.LaunchDescription([
  launch.actions.DeclareLaunchArgument(name='gui', default_value='True',\
    description='Flag to enable joint_state_publisher_gui'),
  launch.actions.DeclareLaunchArgument(name='model',\
    default_value=default_model_path,\
    description='Absolute path to robot urdf file'),
  launch.actions.DeclareLaunchArgument(name='rvizconfig',\
    default_value=default_rviz_config_path,\
    description='Absolute path to rviz config file'),
  joint_state_publisher_node,
  joint_state_publisher_gui_node,
  robot_state_publisher_node,
  rviz_node
])
```

To visualize the result

```
$ colcon build
$ . install/setup.bash
$ ros2 launch sam_bot_description display.launch.py
```





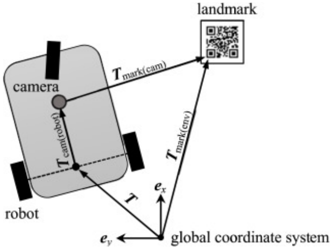
We have successfully created a simple differential drive robot and visualized it in Rviz. Another window was launched - this is a GUI for the joint state publisher. You can manipulate this publisher through the small GUI and the new pose of the joints will be reflected in Rviz.

Odometry

Odometry can come from many sources including LIDAR, RADAR, wheel encoders, VIO, and IMUs. The goal of the odometry is to provide a smooth and continuous local frame based on robot motion. The global positioning system will update the transformation relative to the global frame to account for the odometric drift

It is the role of the **odometry** system to provide the `odom -> base_link` transformation

The **Robot Localization** package is typically used for this fusion. It will take in N sensors of various types and provide a continuous and smooth odometry to TF and to a topic. A typical mobile robotics setup may have odometry from wheel encoders, IMUs, and vision fused in this manor



Setting Up Odometry

Setting up the odometry system for Nav2 for your physical robot depends a lot on which odometry sensors are available with your robot

Let us consider a robot with wheel encoders

To calculate the odometry information, a piece of code that translates wheel encoder information into odometry information is needed:

```
linear = (right_wheel_est_vel + left_wheel_est_vel) / 2;
angular = (right_wheel_est_vel - left_wheel_est_vel) / wheel_separation;
```

The `ros2_control` framework contains various packages for real-time control of robots in ROS 2

The `diff_drive_controller` takes in the `geometry_msgs/Twist` messages published on `cmd_vel` topic, computes odometry information, and publishes `nav_msgs/Odometry` messages on `odom` topic

```
mobile_base_controller:
  type: "diff_drive_controller/DiffDriveController"
  left_wheel: 'wheel_left_joint'
  right_wheel: 'wheel_right_joint'
  pose_covariance_diagonal: [0.01, 0.01, 1000.0, 1000.0, 1000.0, 10.0]
  twist_covariance_diagonal: [0.01, 0.01, 1000.0, 1000.0, 1000.0, 10.0]
```

Setting Up IMU Plugin

```
<link name="imu_link">
  <visual> <geometry>
      <box size="0.1 0.1 0.1"/>
    </geometry> </visual>
  <collision> <geometry>
      <box size="0.1 0.1 0.1"/>
    </geometry> </collision>
  <xacro:box_inertia m="0.1" w="0.1" d="0.1" h="0.1"/>
</link>

<joint name="imu_joint" type="fixed">
  <parent link="base_link"/>
  <child link="imu_link"/>
  <origin xyz="0 0 0.01"/>
</joint>

 <gazebo reference="imu_link">
  <sensor name="imu_sensor" type="imu">
   <plugin filename="libgazebo_ros_imu_sensor.so" name="imu_plugin">
      <ros> <namespace>/demo</namespace>
        <remapping>~/out:=imu</remapping> </ros>
      <initial_orientation_as_reference>
          false
      </initial_orientation_as_reference>
```

Setting Up Differential Drive Plugin

```xml
<gazebo>
  <plugin name='diff_drive' filename='libgazebo_ros_diff_drive.so'>
    <ros> <namespace>/demo</namespace> </ros>
    <!-- wheels -->
    <left_joint>drivewhl_l_joint</left_joint>
    <right_joint>drivewhl_r_joint</right_joint>

    <!-- kinematics -->
    <wheel_separation>0.4</wheel_separation>
    <wheel_diameter>0.2</wheel_diameter>

    <!-- limits -->
    <max_wheel_torque>20</max_wheel_torque>
    <max_wheel_acceleration>1.0</max_wheel_acceleration>

    <!-- output -->
    <publish_odom>true</publish_odom>
    <publish_odom_tf>false</publish_odom_tf>
    <publish_wheel_tf>true</publish_wheel_tf>

    <odometry_frame>odom</odometry_frame>
    <robot_base_frame>base_link</robot_base_frame>
  </plugin>
</gazebo>
```

Launch Gazebo Simulation

To launch the gazebo simulation

```
$ colcon build
$ . install/setup.bash
$ ros2 launch sam_bot_description simulate.launch.py
```

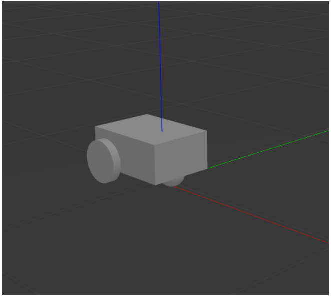
To see the active topics in the system, open a new terminal and execute:

```
$ ros2 topic list
```

You should see `/demo/imu` and `/demo/odom` in the list of topics

To see more information about the topics, execute:

```
$ ros2 topic info /demo/imu
$ ros2 topic info /demo/odom
```



Configuring Robot Localization

The `robot_localization` package provides an Extended Kalman Filter (`ekf_node`) to fuse odometry information and publish the `odom => base_link` transform

First, install the package

```
$ sudo apt install ros-foxy-robot-localization
```

Configuring Robot Localization (Cont.)

Next, create a directory named `config` at the root of your project and create a file named `ekf.yaml`

```
### ekf config file ###
ekf_filter_node:
    ros__parameters:
        frequency: 30.0
        two_d_mode: false
        publish_acceleration: true
        publish_tf: true

        map_frame: map # Defaults to "map" if unspecified
        odom_frame: odom # Defaults to "odom" if unspecified
        base_link_frame: base_link # Defaults to "base_link" ifunspecified
        world_frame: odom # Defaults to the value

        odom0: demo/odom
        odom0_config: [true, true, true, false, false, false, false, false,
                       false, false, false, true, false, false, false]

        imu0: demo/imu
        imu0_config: [false, false, false, true, true, true, false, false,
                      false, false, false, false, false, false, false]
```

Launch and Build Files

Now, let us add the `ekf_node` into a new launch file `localization.launch.py`, then build and run the package

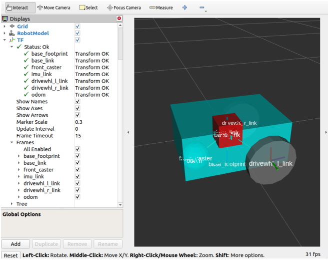
```
$ colcon build
$ . install/setup.bash
$ ros2 launch sam_bot_description localization.launch.py
```

Verify that the `/odometry/filtered`, `/accel/filtered`, and `/tf` topics are active

```
$ ros2 topic list
```

To verify that the `ekf_filter_node` is the subscriber of `/demo/imu` and `/demo/odom` topics

```
$ ros2 topic info /demo/imu
$ ros2 topic info /demo/odom
$ ros2 node info /ekf_filter_node
```

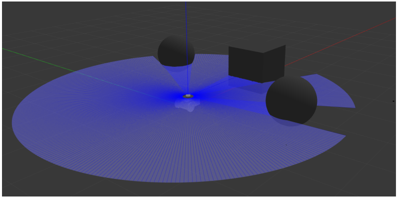


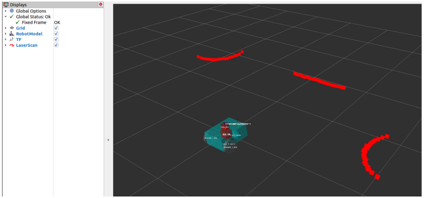
To verify that `robot_localization` is publishing the `odom => base_link` transform

```
$ ros2 run tf2_ros tf2_echo odom base_link
```

sensor_msgs/LaserScan

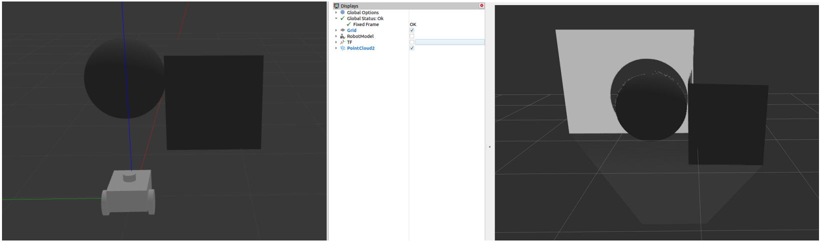
This message represents a single scan from a planar laser range-finder. This message is used in `slam_toolbox` and `nav2_amcl` for localization and mapping, or in `nav2_costmap_2d` for perception





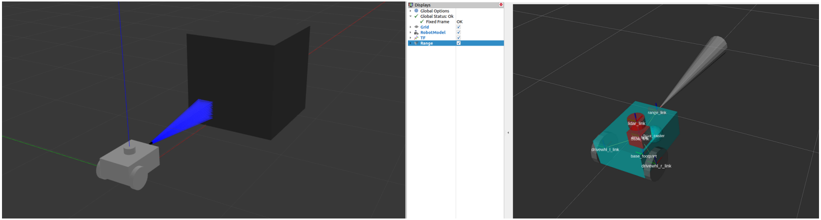
sensor_msgs/PointCloud2

This message holds a collection of 3D points, plus optional additional information about each point. This can be from a 3D lidar, a 2D lidar, a depth camera or more



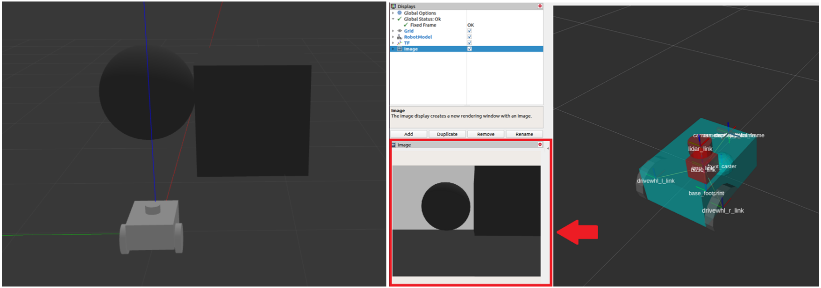
sensor_msgs/Range

This is a single range reading from an active ranger that emits energy and reports one range reading that is valid along an arc at the distance measured. A sonar, IR sensor, or 1D range finder are examples of sensors that use this message



sensor_msgs/Image

This represents the sensor readings from RGB or depth camera, corresponding to RGB or range values

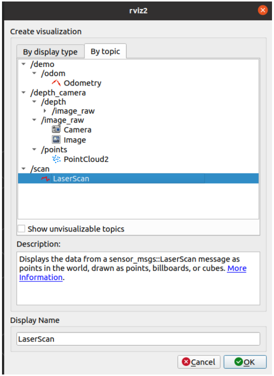


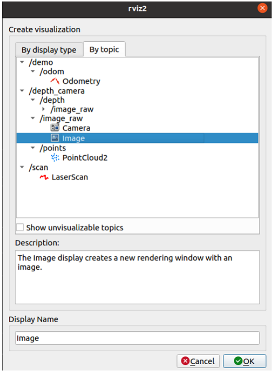
Running Sensor Demo

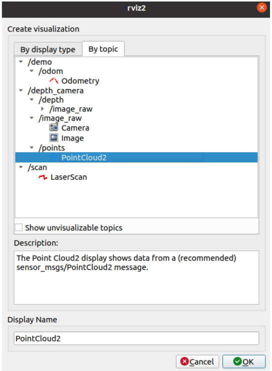
To run the sensor demo

```
$ ros2 launch sam_bot_description sensors.launch.py
```

Add rviz visualization for `LaserScan`, `Image` and `PointCloud2`







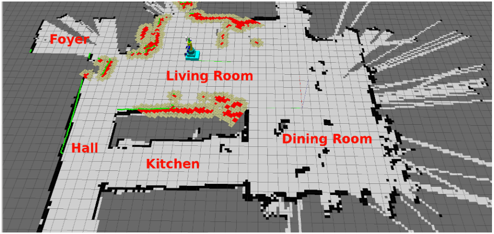
Global Positioning: Localization and SLAM

It is the job of the global positioning system (GPS, SLAM, Motion Capture) to, at minimum, provide the `map -> odom` transformation

The `amcl` package provides an **Adaptive Monte-Carlo Localization** technique based on a particle filter for localization of a static map

The **SLAM Toolbox** is used as the default SLAM algorithm for use to position and generate a static map

These methods may also produce other output including position topics, maps, or other metadata, but they must provide that transformation to be valid. Multiple positioning methods can be fused together using robot localization



SLAM Toolbox

SLAM Toolbox is a set of tools and capabilities for 2D SLAM

- Ordinary point-and-shoot 2D SLAM (start, map, save pgm file)
- Continuing to refine, remap, or continue mapping a saved (serialized) pose-graph at any time
- Life-long mapping: load a saved pose-graph continue mapping in a space while removing extraneous information from new scans
- Optimization-based localization mode built on the pose-graph
- Synchronous and asynchronous modes of mapping
- Kinematic map merging
- Plugin-based optimization solvers with **Google Ceres** plugin
- RVIZ plugin for interacting with the tools
- Graph manipulation tools in RVIZ to manipulate nodes and connections during mapping
- Map serialization and lossless data storage
- ...

Complete documentation at
`https://github.com/SteveMacenski/slam_toolbox`

Launching SLAM Toolbox

Make sure that you have installed the `slam_toolbox` package

```
$ sudo apt install ros-foxy-slam-toolbox
```

The `async_slam_toolbox_node` of the `slam_toolbox` package will do the mapping

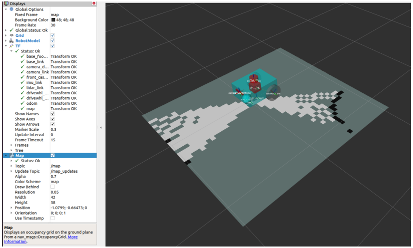
```
$ ros2 launch sam_bot_description slam.launch.py
```

The `slam_toolbox` should now be publishing to the `/map` topic and providing the `map => odom` transform

We can also check that the transforms are correct by executing the command:

```
$ ros2 run tf2_tools view_frames.py
```

The line above will create a `frames.pdf` file that shows the current transform tree



Configuring the Robot's Footprint

Under the config directory,open the file named `nav2_params.yaml`

The footprint parameter of the local costmap is set with a rectangular-shaped footprint.

This box is centered at the `base_link` frame of `sam_bot`

```
resolution: 0.05
footprint: "[[0.21, 0.195],[0.21, -0.195],[-0.21, -0.195],[-0.21, 0.195]]"
plugins: ["voxel_layer", "inflation_layer"]
```

For the global costmap, the `robot_radius` parameter is set to create a circular footprint that matches `sam_bot`'s size and centered at `base_link`

```
use_sim_time: True
robot_radius: 0.3
resolution: 0.05
```

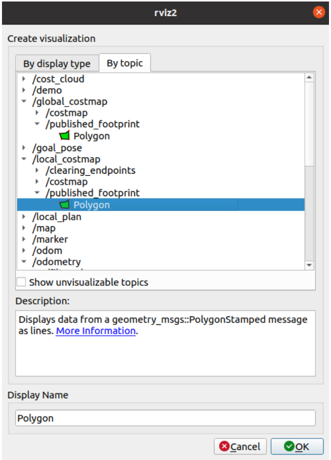
Visualizing the Robot's Footprint

First, build and run the project

```
$ colcon build
$ . install/setup.bash
$ ros2 launch sam_bot_description navigation.launch.py
```

To visualize the footprint of the local costmap in RViz, windowclick "Add" and under the "By topic" tab, select the Polygon under the `/local_costmap/published_footprint` topic

The same could be for the `/global_costmap/published_footprint` topic



Setting Up the Planner

The **planner server** may utilize plugins that work on the map space

- The `NavFn Planner` is a navigation function planner that uses either Dijkstra or A*
- The `Smac 2D Planner` implements a 2D A* algorithm using 4 or 8 connected neighborhoods with a smoother and multi-resolution query
- The `Theta Star Planner` is an implementation of Theta* using either line of sight to create non-discretely oriented path segments
- The `Smac Hybrid-A* Planner` that supports arbitrary shaped ackermann and legged robots. It is a highly optimized and fully reconfigurable Hybrid-A* implementation supporting Dubin and Reeds-Shepp motion models, considering the robot's minimum turning radius constraint and the robot's full footprint for collision avoidance
- The `Smac Lattice Planner` that expanding the robot state space while ensuring the path complies with the robot's kinematic constraints, providing minimum control sets which allows it to support differential, omnidirectional, and ackermann vehicles

Planner Configuration

Plugin Name	Supported Robot Types
NavFn Planner	Circular Differential, Circular Omnidirectional
Smac Planner 2D	
Theta Star Planner	
Smac Hybrid-A* Planner	Non-circular or Circular Ackermann, Non-circular or Circular Legged
Smac Lattice Planner	Non-circular Differential, Non-circular Omnidirectional

Configuration example

```
planner_server:
  ros__parameters:
    planner_plugins: ['GridBased']
    GridBased:
      plugin: 'nav2_navfn_planner/NavfnPlanner'
```

Setting Up the Controller

The **Controller Server** can use different plugins to steer the robot along the path

- The `DWB Controller` implements a modified Dynamic Window Approach (DWA) algorithm with configurable plugins to compute the control commands for the robot
- The `TEB Controller` is an MPC time optimal controller implementing the Timed Elastic Band (TEB) approach which optimizes the robot's trajectory based on its execution time, distance from obstacles, and feasibility with respect to the robot's kinematic constraints
- The `Regulated Pure Pursuit controller` (RPP) implements a variant of the pure pursuit algorithm with added regulation heuristic functions to manage collision and velocity constraints

All of these controllers work for both circular and non-circular robots

Controller Configuration

Plugin Name	Supported Robot Types	Task
DWB	Differential, Omnidirectional	Dynamic obstacle avoidance
TEB	Differential, Omnidirectional, Ackermann, Legged	
RPP	Differential, Ackermann, Legged	Exact path following

Configuration example

```
controller_server:
  ros__parameters:
    controller_plugins: ["FollowPath"]
    FollowPath:
       plugin: "dwb_core::DWBLocalPlanner"
```

Nav2 Behavior Trees

The following behavior tree plans a new path to goal every 1 meter (set by `DistanceController`) using `ComputePathToPose`

`FollowPath` will take this path and follow it using the default algorithm

```xml
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <PipelineSequence name="NavigateWithReplanning">
      <DistanceController distance="1.0">
        <ComputePathToPose goal="{goal}" path="{path}"/>
      </DistanceController>
      <FollowPath path="{path}"/>
    </PipelineSequence>
  </BehaviorTree>
</root>
```

This tree contains:

- No recovery methods
- No retries on failure
- No selectionof the planner or controller algorithms
- No integration with automatic door, elevator, or other APIs
- No subtrees for other behaviors like docking, following, etc.

Nav2 Action Nodes

Action Nodes return

- `SUCCESS` if the action server believes the action has been completed correctly
- `RUNNING` when the node is still running
- `FAILURE` otherwise

Predefined **Action Nodes**

- ComputePathToPose - ComputePathToPose Action Server Client (Planner Interface)
- FollowPath - FollowPath Action Server Client (Controller Interface)
- Spin, Wait, Backup - Recoveries Action Server Client
- ClearCostmapService - ClearCostmapService Server Clients

Nav2 Condition Nodes

Condition Nodes return

- `SUCCESS` if the condition is `TRUE`
- and `FAILURE` if the condition is `FALSE`

Predefined **Condition Nodes**

- GoalUpdated - Checks if the goal on the goal topic has been updated
- GoalReached - Checks if the goal has been reached
- InitialPoseReceived - Checks to see if a pose on the `intial_pose` topic has been received
- isBatteryLow - Checks to see if the battery is low by listening on the battery topic

Nav2 Decorator Nodes

Decorator Nodes

- Distance Controller - Will tick children nodes every time the robot has traveled a certain distance
- Rate Controller - Controls the ticking of it's child node at a constant frequency. The tick rate is an exposed port
- Goal Updater - Will update the goal of children nodes via ports on the BT
- Single Trigger - Will only tick it's child node once, and will return `FAILURE` for all subsequent ticks
- Speed Controller - Controls the ticking of it's child node at a rate proportional to the robot's speed

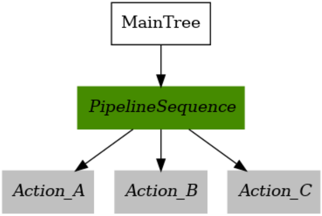
Nav2 Control Nodes: PipelineSequence

The **PipelineSequence** control node

- Re-ticks previous children when a child returns RUNNING
- If at any point a child returns FAILURE, all children will be halted and the parent node will also return FAILURE
- Upon SUCCESS of the last node in the sequence, this node will halt and return SUCCESS

Example:

```xml
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <PipelineSequence>
            <Action_A/>
            <Action_B/>
            <Action_C/>
        </PipelineSequence>
    </BehaviorTree>
</root>
```



1) A, B, and C are all IDLE

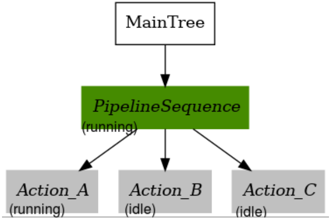
Nav2 Control Nodes: PipelineSequence

The **PipelineSequence** control node

- Re-ticks previous children when a child returns `RUNNING`
- If at any point a child returns `FAILURE`, all children will be halted and the parent node will also return `FAILURE`
- Upon `SUCCESS` of the last node in the sequence, this node will halt and return `SUCCESS`

Example:

```xml
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <PipelineSequence>
            <Action_A/>
            <Action_B/>
            <Action_C/>
        </PipelineSequence>
    </BehaviorTree>
</root>
```



2) When the parent PipelineSequence is first ticked, let's assume A returns RUNNING. The parent node will now return RUNNING and no other nodes are ticked

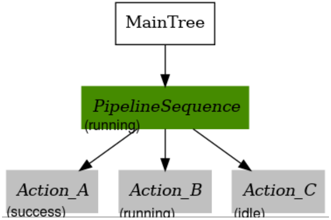
Nav2 Control Nodes: PipelineSequence

The **PipelineSequence** control node

- Re-ticks previous children when a child returns RUNNING
- If at any point a child returns FAILURE, all children will be halted and the parent node will also return FAILURE
- Upon SUCCESS of the last node in the sequence, this node will halt and return SUCCESS

Example:

```xml
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <PipelineSequence>
            <Action_A/>
            <Action_B/>
            <Action_C/>
        </PipelineSequence>
    </BehaviorTree>
</root>
```



3) Now, let's assume A returns SUCCESS, B will now get ticked and will return RUNNING. C has not yet been ticked so will return IDLE

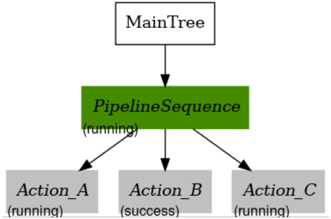
Nav2 Control Nodes: PipelineSequence

The **PipelineSequence** control node

- Re-ticks previous children when a child returns `RUNNING`
- If at any point a child returns `FAILURE`, all children will be halted and the parent node will also return `FAILURE`
- Upon `SUCCESS` of the last node in the sequence, this node will halt and return `SUCCESS`

Example:

```xml
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <PipelineSequence>
            <Action_A/>
            <Action_B/>
            <Action_C/>
        </PipelineSequence>
    </BehaviorTree>
</root>
```



4) A gets ticked again and returns RUNNING, and B gets re-ticked and returns SUCCESS and therefore the BT goes on to tick C for the first time and returns RUNNING

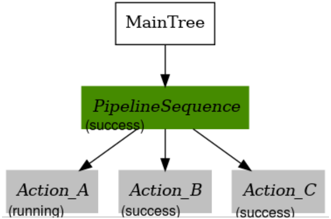
Nav2 Control Nodes: PipelineSequence

The **PipelineSequence** control node

- Re-ticks previous children when a child returns `RUNNING`
- If at any point a child returns `FAILURE`, all children will be halted and the parent node will also return `FAILURE`
- Upon `SUCCESS` of the last node in the sequence, this node will halt and return `SUCCESS`

Example:

```xml
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <PipelineSequence>
            <Action_A/>
            <Action_B/>
            <Action_C/>
        </PipelineSequence>
    </BehaviorTree>
</root>
```



5) Let's assume A is still RUNNING, B returns SUCCESS, and C now returns SUCCESS. The sequence is now complete, and therefore A is halted, even though it was still RUNNING

Nav2 Control Nodes: Recovery

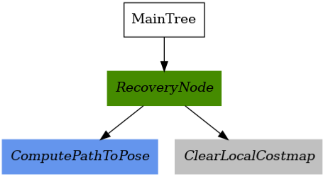
The **Recovery** control node has only two children

- It returns SUCCESS if and only if the first child returns SUCCESS
- If the first child returns FAILURE, the second child will be ticked

This loop will continue until either:

- The first child returns SUCCESS
- The second child returns FAILURE, which results in FAILURE of the parent node
- The number_of_retries input parameter is violated

```xml
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <RecoveryNode number_of_retries="1">
            <ComputePathToPose/>
            <ClearLocalCostmap/>
        </RecoveryNode>
    </BehaviorTree>
</root>
```



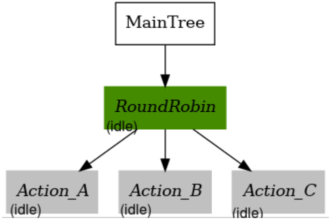
Nav2 Control Nodes: RoundRobin

The **RoundRobin** control node ticks it's children in a round robin fashion until

- A child returns `SUCCESS`, in which the parent node returns `SUCCESS`
- If all children return `FAILURE`, the parent returns `FAILURE`

Example:

```
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <RoundRobin>
            <Action_A/>
            <Action_B/>
            <Action_C/>
        </RoundRobin>
    </BehaviorTree>
</root>
```



1) A, B, and C are all IDLE

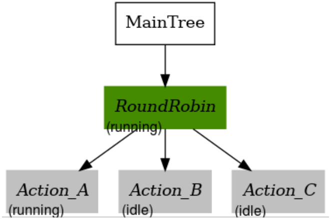
Nav2 Control Nodes: RoundRobin

The **RoundRobin** control node ticks it's children in a round robin fashion until

- A child returns SUCCESS, in which the parent node returns SUCCESS
- If all children return FAILURE, the parent returns FAILURE

Example:

```
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <RoundRobin>
            <Action_A/>
            <Action_B/>
            <Action_C/>
        </RoundRobin>
    </BehaviorTree>
</root>
```



2) Upon tick of the parent node, the first child is ticked

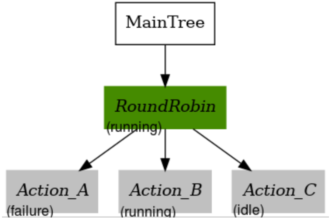
Nav2 Control Nodes: RoundRobin

The **RoundRobin** control node ticks it's children in a round robin fashion until

- A child returns `SUCCESS`, in which the parent node returns `SUCCESS`
- If all children return `FAILURE`, the parent returns `FAILURE`

Example:

```xml
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <RoundRobin>
            <Action_A/>
            <Action_B/>
            <Action_C/>
        </RoundRobin>
    </BehaviorTree>
</root>
```



3) Let's assume that A returns FAILURE. B will get ticked next, and C remains unticked

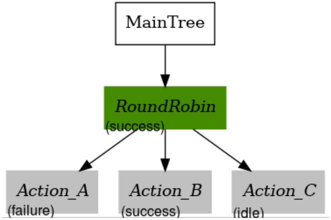
Nav2 Control Nodes: RoundRobin

The **RoundRobin** control node ticks it's children in a round robin fashion until

- A child returns `SUCCESS`, in which the parent node returns `SUCCESS`
- If all children return `FAILURE`, the parent returns `FAILURE`

Example:

```xml
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <RoundRobin>
            <Action_A/>
            <Action_B/>
            <Action_C/>
        </RoundRobin>
    </BehaviorTree>
</root>
```



4) B returns SUCCESS. The parent RoundRobin will now halt all children and returns SUCCESS. The parent node ticks C upon the next tick rather than start from A

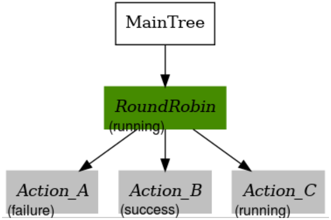
Nav2 Control Nodes: RoundRobin

The **RoundRobin** control node ticks it's children in a round robin fashion until

- A child returns `SUCCESS`, in which the parent node returns `SUCCESS`
- If all children return `FAILURE`, the parent returns `FAILURE`

Example:

```
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <RoundRobin>
            <Action_A/>
            <Action_B/>
            <Action_C/>
        </RoundRobin>
    </BehaviorTree>
</root>
```



5) On this tick, let's assume C returns RUNNING, and so does the parent RoundRobin. No other nodes are ticked

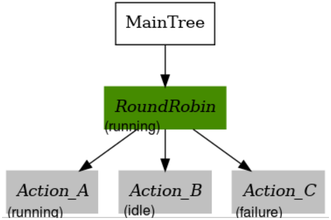
Nav2 Control Nodes: RoundRobin

The **RoundRobin** control node ticks it's children in a round robin fashion until

- A child returns SUCCESS, in which the parent node returns SUCCESS
- If all children return FAILURE, the parent returns FAILURE

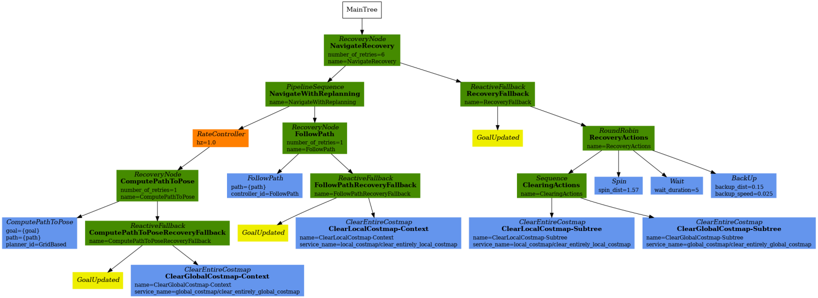
Example:

```
<root main_tree_to_execute="MainTree">
    <BehaviorTree ID="MainTree">
        <RoundRobin>
            <Action_A/>
            <Action_B/>
            <Action_C/>
        </RoundRobin>
    </BehaviorTree>
</root>
```



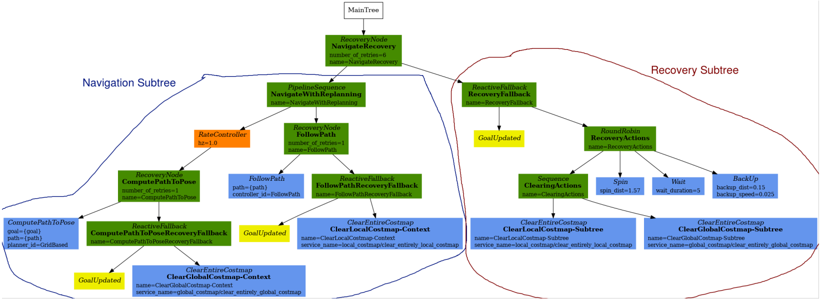
6) Let's assume C returns FAILURE. The parent will tick A again. A returns RUNNING and so will the parent RoundRobin node. This pattern will continue indefinitely unless all children return FAILURE.

Navigate To Pose With Replanning and Recovery



This tree can be broken into two smaller subtrees that we can focus on one at a time. These smaller subtrees are the children of the top-most `RecoveryNode`

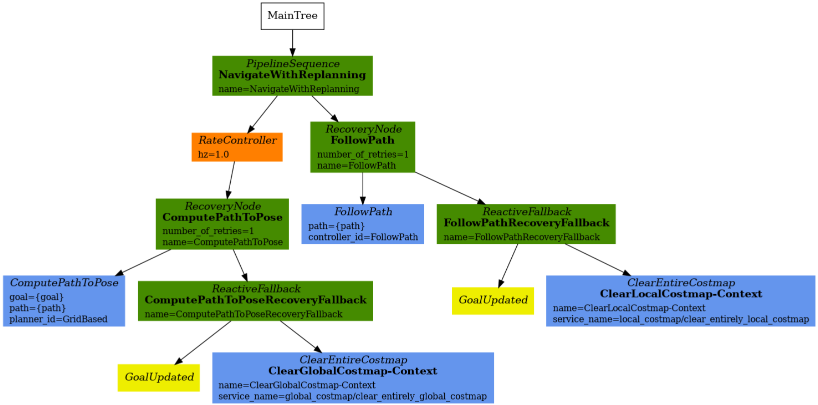
Navigate To Pose With Replanning and Recovery



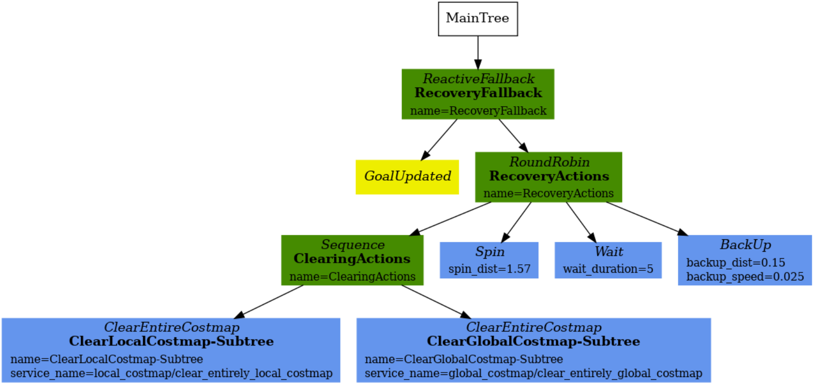
This tree can be broken into two smaller subtrees that we can focus on one at a time. These smaller subtrees are the children of the top-most `RecoveryNode`

From this point forward the `NavigateWithReplanning|` `subtree will be referred to as the Navigation|` `subtree, and the RecoveryFallb` `subtree will be known as the Recovery|` `subtree`

Navigation Subtree



Recovery Subtree



References

- ROS2 Tutorials
  https://docs.ros.org/en/foxy/Tutorials.html
- Turtlebot3 e-manuals
  https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/
- SLAM Toolbox Documentation and Source Code
  https://github.com/SteveMacenski/slam_toolbox
- ROS Navigation 2 Tutorials
  https://navigation.ros.org/index.html
- ROS Navigation 2 Documenation and Source Code
  https://github.com/ros-planning/navigation2
- ROS Behavoir Tree Documentation
  https://index.ros.org/p/nav2_bt_navigator/
- Behavior Tree Main Page https://www.behaviortree.dev/

Thanks!

Questions?

The only stupid question is the one you were afraid to ask but never did.
-Rich Sutton