Navigation Concepts

This page is to help familiarize new robotists to the concepts of mobile robot navigation, in particular, with the concepts required to appreciating and working with this project.

ROS 2

ROS 2 is the core middleware used for Nav2. If you are unfamilar with this, please visit the ROS 2 documentation before continuing.

Action Server

Just as in ROS, action servers are a common way to control long running tasks like navigation. This stack makes more extensive use of actions, and in some cases, without an easy topic interface. It is more important to understand action servers as a developer in ROS 2. Some simple CLI examples can be found in the ROS 2 documentation.

Action servers are similar to a canonical service server. A client will request some task to be completed, except, this task may take a long time. An example would be moving the shovel up from a bulldozer or ask a robot to travel 10 meters to the right.

In this situation, action servers and clients allow us to call a long-running task in another process or thread and return a future to its result. It is permissible at this point to block until the action is complete, however, you may want to occasionally check if the action is complete and continue to process work in the client thread. Since it is long-running, action servers will also provide feedback to their clients. This feedback can be anything and is defined in the ROS `.action` along with the request and result types. In the bulldozer example, a request may be an angle, a feedback may be the angle remaining to be moved, and the result is a success or fail boolean with the end angle. In the navigation example, a request may be a position, a feedback may be the time its been navigating for and the distance to the goal, and the result a boolean for success.

Feedback and results can be gathered synchronously by registering callbacks with the action client. They may also be gathered by asychronously requesting information from the shared future objects. Both require spinning the client node to process callback groups.

Action servers are used in this stack to communicate with the highest level 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 and Bond

Lifecycle (or Managed, more correctly) nodes are unique to ROS 2. More information can be found here. They are nodes that 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.

When a node is started, it is in the unconfigured state, only processing the node's constructor which should **not** contain any ROS networking setup or parameter reading. By the launch system, or the supplied lifecycle manager, the nodes need to be transitioned to inactive by configuring. After, it is possible to activate the node by transitioning through the activating stage.

This state will allow the node to process information and be fully setup to run. The configuration stage, triggering the `on_configure()` method, will setup all parameters, ROS networking interfaces, and for safety systems, all dynamically allocated memory. The activation stage, triggering the `on_activate()` method, will active the ROS networking interfaces and set any states in the program to start processing information.

To shutdown, we transition into deactivating, cleaning up, shutting down and end in the finalized state. The networking interfaces are deactivated and stop processing, deallocate memory, exit cleanly, in those stages, respectively.

The lifecycle node framework is used extensively through out this project and all servers utilize it. It is best convention for all ROS systems to use lifecycle nodes if it is possible.

Within Nav2, we use a wrapper of LifecycleNodes, `nav2_util LifecycleNode`. This wrapper wraps much of the complexities of LifecycleNodes for typical applications. It also includes a `bond` connection for the lifecycle manager to ensure that after a server transitions up, it also remains active. If a server crashes, it lets the lifecycle manager know and transition down the system to prevent a critical failure. See Eloquent to Foxy for details.

Behavior Trees

Behavior trees (BT) are becoming increasingly common in complex robotics tasks. They 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. An example would be a soccer playing robot. Embedding the logic of soccer game play into a FSM would be challenging and error prone with many possible states and rules. Additionally, modeling choices like to shoot at the goal from the left, right, or center, is particularly unclear. With a BT, basic primitives like "kick" "walk" "go to ball" can be created and reused for many behaviors. More information can be found in this book. I **strongly** recommend reading chapters 1-3 to get a good understanding of the nomenclature and workflow. It should only take about 30 minutes.

For this project, we use BehaviorTree CPP V3 as the behavior tree library. We create node plugins which can be constructed into a tree, inside the `BT Navigator`. The node plugins are loaded into the BT and when the XML file of the tree is parsed, the registered names are associated. At this point, we can march through the behavior tree to navigate.

One reason this library is used is its ability to load subtrees. This means that the Nav2 behavior tree can be loaded into another higher-level BT to use this project as node plugin. An example would be in soccer play, using the Nav2 behavior tree as the "go to ball" node with a ball detection as part of a larger task. Additionally, we supply a `NavigateToPoseAction` plugin for BT so the Nav2 stack can be called from a client application through the usual action interface.

Navigation Servers

Planners and controllers are at the heart of a navigation task. Recoveries are used to get the robot out of a bad situation or attempt to deal with various forms of issues to make the system fault-tolerant. In this section, the general concepts around them and their uses in this project are analyzed.

Planner, Controller, and Recovery Servers

Three of the action servers in this project are the planner, recovery, and controller servers. These action servers are used to host a map of algorithm plugins to complete various tasks. They also host the environmental representation used by the algorithm plugins to compute their outputs.

The planner and controller servers will be configured at runtime with the names (aliases) and types of algorithms to use. These types are the pluginlib names that have been registered and the names are the aliases for the task. An example would be the DWB controller used with name `FollowPath`, as it follows a reference path. In this case, then all parameters for DWB would be placed in that namespace, e.g. `FollowPath.<param>`.

These two servers then expose an action interface corresponding to its task. When the behavior tree ticks the corresponding BT node, it will call the action server to process its task. The action server callback inside the server will call the chosen algorithm by its name (e.g. `FollowPath`) that maps to a specific algorithm. This allows a user to abstract the algorithm used in the behavior tree to classes of algorithms. For instance, you can have `N` plugin controllers to follow paths, dock with charger, avoid dynamic obstacles, or interface with a tool. Having all of these plugins in the same server allows the user to make use of a single environmental representation object, which is costly to duplicate.

For the recovery server, each of the recoveries also contains their own name, however, each plugin will also expose its own special action server. This is done because of the wide variety of recovery actions that may be created cannot have a single simple interface to share. The recovery server also contains a costmap subscriber to the local costmap, receiving real-time updates from the controller server, to compute its tasks. We do this to avoid having multiple instances of the local costmap which are computationally expensive to duplicate.

Alternatively, since the BT nodes are trivial plugins calling an action, new BT nodes can be created to call other action servers with other action types. It is advisable to use the provided servers if possible at all times. If, due to the plugin or action interfaces, a new server is needed, that can be sustained with the framework. The new server should use the new type and plugin interface, similar to the provided servers. A new BT node plugin will need to be created to call the new action server – however no forking or modification is required in the Nav2 repo itself by making extensive use of servers and plugins.

If you find that you require a new interface to the pluginlib definition or action type, please file a ticket and see if we can rectify that in the same interfaces.

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.

Controllers

Controllers, also known as local planners in ROS 1, 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.

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.

Waypoint Following

Waypoint following is a basic feature of a navigation system. It tells our system how to use navigation to get to multiple destinations.

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 fully sufficient to create a production-grade on-robot solution. Since the autonomy system / dispatcher is taking into account things like the robot's pose, battery level, current task, and more when assigning tasks, 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 this situation, you should think of a request to the waypoint follower as 1 unit of work (e.g. 1 pick in a warehouse, 1 security patrole loop, 1 aisle, etc) to do a task and then return to the dispatcher for the next task or request to recharge. In this school of thought, the waypoint following application is just one step above navigation and below the system autonomy application.

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. In this case, you should use the `nav2_behavior_tree` package to create a custom application-level behavior tree using navigation to complete the task. This can include subtrees like checking for the charge status mid-task for returning to dock or handling more than 1 unit of work in a more complex task. Soon, there will be a `nav2_bt_waypoint_follower` (name subject to adjustment) that will allow you to create this application more easily. In this school of thought, the waypoint following application is more closely tied to the system autonomy, or in many cases, is the system autonomy.

Neither is better than the other, it highly depends on the tasks your robot(s) are completing, in what type of environment, and with what cloud resources available. Often this distinction is very clear for a given business case.

State Estimation

Within the navigation project, there are 2 major transformations that need to be provided, according to community standards. The `map` to `odom` transform is provided by a positioning system (localization, mapping, SLAM) and `odom` to `base_link` by an odometry system.

Note

There is **no** requirement on using a LIDAR on your robot to use the navigation system. There is no requirement to use lidar-based collision avoidance, localization, or slam. However, we do provide instructions and support tried and true implementations of these things using lidars. You can be equally as successful using a vision or depth based positioning system and using other sensors for collision avoidance. The only requirement is that you follow the standards below with your choice of implementation.

Standards

REP 105 defines the frames and conventions required for navigation and the larger ROS ecosystem. These conventions should be followed at all times to make use of the rich positioning, odometry, and slam projects available in the community.

In a nutshell, REP-105 says that you must, at minimum, build a TF tree that contains a full `map` -> `odom` -> `base_link` -> `[sensor frames]` for your robot. TF2 are the time-variant transformation library in ROS 2 we use to represent and obtain time synchronized transformations. It is the job of the global positioning system (GPS, SLAM, Motion Capture) to, at minimum, provide the `map` -> `odom` transformation. It is then the role of the odometry system to provide the `odom` -> `base_link` transformation. The remainder of the transformations relative to `base_link` should be static and defined in your URDF.

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. We provide `amcl` which is an Adaptive Monte-Carlo Localization technique based on a particle filter for localization of a static map. We also provide SLAM Toolbox 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, discussed more below.

Odometry

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

Robot Localization 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.

The smooth output can be used then for dead-reckoning for precise motion and updating the position of the robot accurately between global position updates.

Environmental Representation

The environmental representation is the way the robot perceives its environment. It also acts as the central localization for various algorithms and data sources to combine their information into a single space. This space is then used by the controllers, planners, and recoveries to compute their tasks safely and efficiently.

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.

Costmap layers can be created to detect and track obstacles in the scene for collision avoidance using camera or depth sensors. Additionally, layers can be created to algorithmically change the underlying costmap based on some rule or heuristic. Finally, they may be used to buffer live data into the 2D or 3D world for binary obstacle marking.

Costmap Filters

Imagine, you're annotating a map file (or any image file) in order to have a specific action occur based on the location in the annotated map. Examples of marking/annotating might be keep out zones to avoid planning inside, or have pixels belong to maximum speeds in marked areas. This annotated map is called "filter mask". Just like a mask overlaid on a surface, it can or cannot be same size, pose and scale as a main map. The main goal of filter mask - is to provide an ability of marking areas on maps with some additional features or behavioral changes.

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. These plugins are called "filters" as they are filtering a costmap by spatial annotations marked on filter masks. In order to make a filtered costmap and change robot's behavior in annotated areas, filter plugin reads the data came from filter mask. This data is being linearly transformed into feature map in a filter space. Having this transformed feature map along with a map/costmap, any sensors data and current robot coordinates filters can update underlying costmap and change behavior of the robot depending on where it is. 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.

Other Forms

Various other forms of environmental representations exist. These include:

- gradient maps, which are similar to costmaps but represent surface gradients to check traversibility over
- 3D costmaps, which represent the space in 3D, but then also requires 3D planning and collision checking
- Mesh maps, which are similar to gradient maps but with surface meshes at many angles
- “Vector space”, taking in sensor information and using machine learning to detect individual items and locations to track rather than buffering discrete points.

Nav2 Academic Overview