Nav2

Stefano Carpin

Department of Computer Science and Engineering
School of Engineering
University of California, Merced
https://sites.ucmerced.edu/scarpin
https//robotics.ucmerced.edu

ROS Actions

- non-blocking, preemptable function calls
  - non-blocking: control returns to the caller
  - pre-emptable: execution can be interrupted before it terminates
- contrast with *services*
- extensively used in the ROS 2 navigation stack (aka Nav2)
- completing some operations (e.g., navigating to a point) may take a long time

Key Concepts

Action Server : the node offering and implementing an action.

Action Client : the node sending a request to an action server.

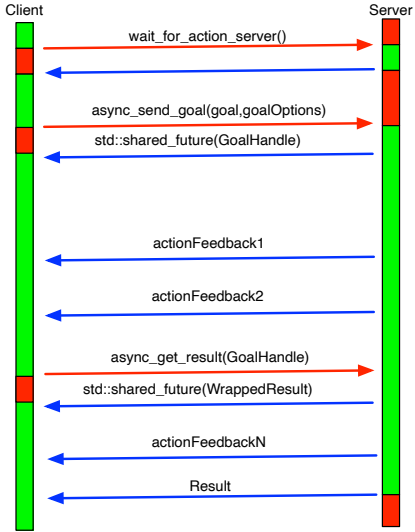
Goal : the message sent by the client to the server when an action is initiated

Feedback : a message periodically sent back by the server to the client

Result : a unique message sent by the server back to the client once the action is completed (e.g, success or failure.)

- busy waiting is also possible (but discouraged)
- support for actions is provided by the package `rclcpp_action`

Logic Flow


Client
Server
wait_for_action_server()
async_send_goal(goal,goalOptions)
std::shared_future(GoalHandle)
actionFeedback1
actionFeedback2
async_get_result(GoalHandle)
std::shared_future(WrappedResult)
actionFeedbackN
Result

Relevant Functions

`wait_for_action_server` : waits until the action server is ready to receive a goal. Can accept a timeout parameter to limit the wait time.

`action_server_is_ready` : checks if the action server is ready to accept a goal request.

`async_send_goal` : sends a goal to the action server and registers callback functions

`async_get_result` : gets the result for an active goal specified through a goal handle returned by `async_send_goal`. The function returns a shared future to a `WrappedResult` holding the result.

`async_cancel_goal` : pre-empts the action by canceling the current goal. The function returns a future that is set when the cancel request is honored.

More about actions

- must provide three callback functions (one for the goal, one for the feedback, one for the result)
- `spin` functions must be called to trigger the callback functions
- actions proceed asynchronously as a separate thread and are built on top of the C++ concurrency support library
- many functions related to actions return a C++ *future*

Futures

- action clients and servers run concurrently
- implemented on top of C++ concurrency library
- results often returned as instances of `std::shared_future`.
- through a future, a client can verify if the server has completed its computation and provided a result
- call `spin_until_future_complete` to spin until the result expected to be passed back through a future is ready (blocking)

Goals, Goal Options, Goal Handles and Wrapped Results

- `async_send_goal` requires a Goal and SendGoalOptions
- Goal information can be obtained with `ros2 interface show`
- SendGoalOptions features three pointers to the callback functions
- `async_send_goal` returns a future to `GoalHandle`
- the future to `GoalHandle` can be passed to `async_get_result` to get a future to `WrappedResult`
- when `WrappedResult` is ready, one can extract the result
- or one can skip this altogether and just do everything via the callback functions

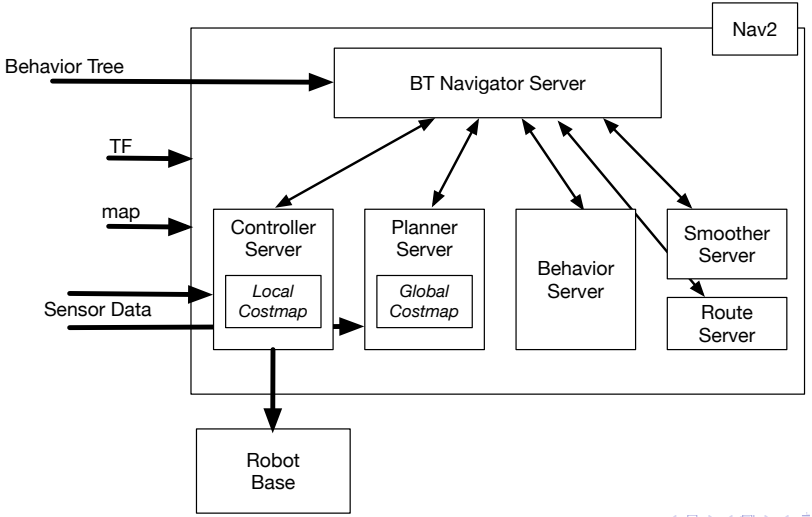
Example: spinning the Turtlebot in place

- action of type `nav2msgs::action::Spin`; inspect its structure with `ros2 interface show`
- proivided by the navigation stack, rotates the robot in place (has nothing to do with the `spin` function
- goal consists of `target_yaw` and `time_allowance` (an instance of `Duration`)
- see `actioncall.cpp` for full example

Planning in ROS

- *Nav2* provides both planning and navigation
- integrates perception and replanning; implements many of the concepts seen so far
- necessitates of various other nodes in place (e.g., to localize the robot, get information about surroundings, etc.)
- does not implement open loop planning, but rather integrates sensors and feedback

Overall Nav2 architecture



Nav2 inputs and outputs

Inputs

- a behavior tree specifying how to orchestrate the interactions among submodules;
- transformations from `tf2`, which describe the kinematic arrangement of objects in the environment (including the current pose of the robot), as well as the configuration of sensors mounted on the robot;
- a map of the environment in which the robot is operating;
- sensor data from the robot's onboard systems, particularly sensors capable of measuring distances to obstacles.

Ouputs: a stream of commands sent to the robot's actuators

Nav2 Servers

Planner: the planner module is in charge of computing a global plan between assigned start and goal poses. The computed plan is informed by the available map of the environment, and by the kinematic model and shape of the robot. The output of the planner is an *open loop* reference path. This module is implemented by the planner server.

Controller: the controller module follows a path computed by the planner (aka local plan). It does so by issuing velocity commands to the underlying platform. In doing so the controller implements a *closed loop* strategy, i.e., it continuously monitors the onboard sensors to ensure the robot follows the assigned path. This module is implemented by the controller server.

Nav2 Servers

Smoother: the smoother modifies a path produced by the planner to account for additional criteria that the planner may have not accounted for. This module is implemented by the smoother server.

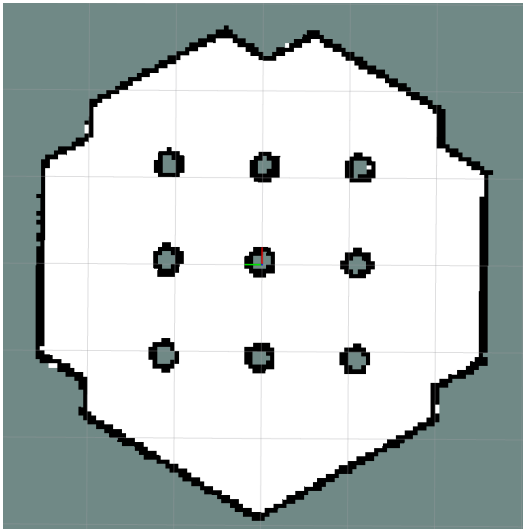
Behavior Server: implements maneuvers to handle failures or unforeseen events (e.g., when the robot gets stuck and cannot make progress). The behaviors are triggered by the BT Navigation Server when the robot fails to make progress toward its assigned goal. Currently, implements three maneuvers: *spin*, *wait*, and *back up*.

Route Server: high-level planner that computes paths on a navigation graph. This differs from the Planner Server, which computes paths based on the environment map, although the navigation graph may have been extracted from the environment map.

Localization, Maps, Cost Maps

- Nav2 must know the current pose of the robot within its environment
- localization is provided by a localization algorithm matching sensor readings to an a priori map of the environment
- ... therefore a map of the environment is needed, too.
- global cost map: built on top of the map; used by the planner server for global planning
- local cost map: dynamically built based on sensor inputs; used by the controller server for local planning
- the global is provided through the `/map` topic, and is an instance of `nav_msgs::msg::OccupancyGrid`

Map



Layers

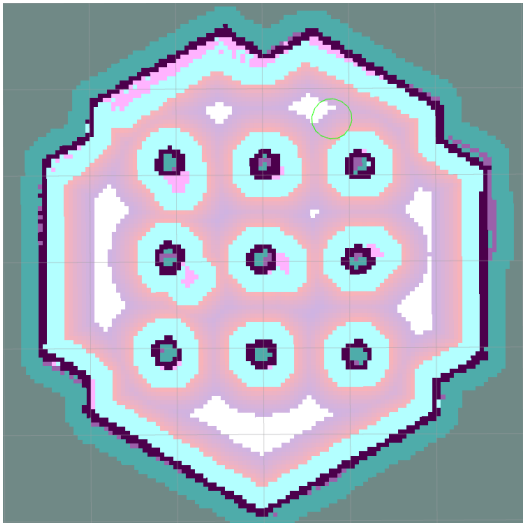
Static Layer: Represents the static map of the environment retrieved from `/map` at startup.

Obstacle Layer: Incorporates information about static and dynamic obstacles detected by the onboard sensors. This layer handles dynamic components that were not modeled in the static layer.

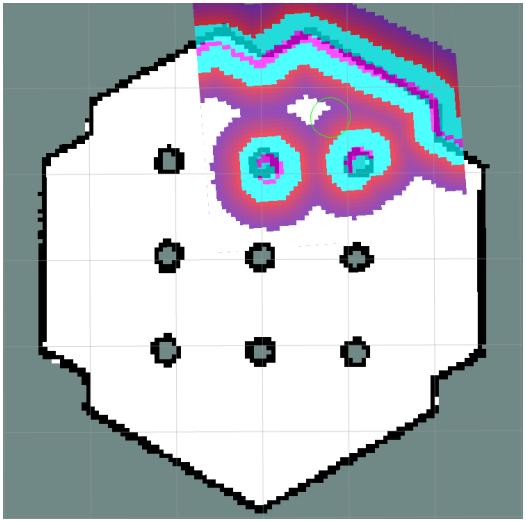
Voxel Layer: Similar to the obstacle layer, but relies on sensors providing three-dimensional data, such as depth cameras or three-dimensional range finders.

Inflation Layer: The inflation layer adds a safety margin by increasing the costs of cells surrounding the obstacles (thus inflating them). Encourages the planner to stay clear of obstacles when possible. The inflation layer retrieves its data from the other layers. The width of the inflation can be configured using a parameter called `inflation_radius`, which should be chosen based on the size of the robot.

Global cost map



Local cost map



Global and local cost map

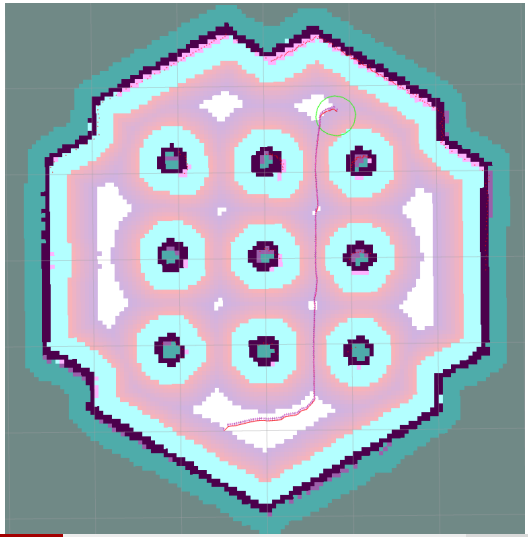
Maps and costmaps

- handled by the `nav2_costmap_2d` package, and they are all instances of `nav_msgs::msg::OccupancyGrid`.
- bidimensional regular grids, where each cell contains a cost value between 0 and 255.
  - 0: free space
  - 253: obstacles
  - 254: *lethal obstacles* (aka forbidden region)
  - 255: unknown
  - Intermediate values are also used, with higher values guiding the planner to prefer paths that maintain a safe clearance from obstacles

Planner Server

- can be linked to many plugins
- we will use the NavfnPlanner plugin
- compute a path from the current cell to a goal cell
- how? build a navigation function assigning costs to cells using the costs in the global cost map
- grid cells (states) are explored using either Dijkstra or A*
- produces a path, but does not send velocity commands to the robot

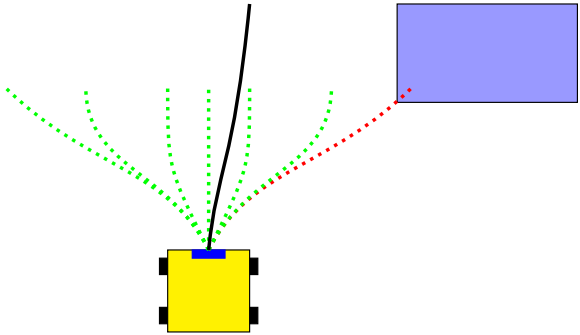
Global cost map



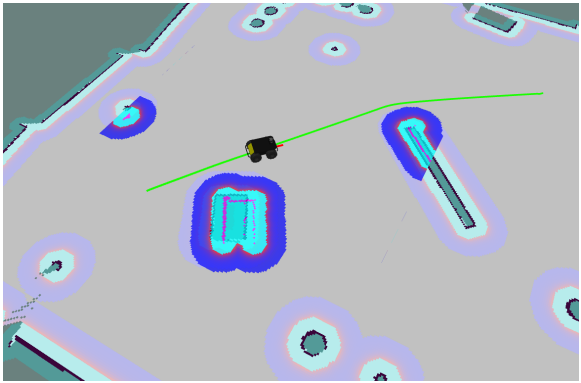
Controller Server

- implemented by the node `controller_server` through the action `follow_path`.
- sends velocity commands to the robot trying to match the global path produced by the planner server (an instance of `nav::msg::Path`)
- in charge of reacting to unforeseen changes and errors (using the local cost map)
- can be linked to many plugins
- we will use the DWB plugin
  - generates set of candidate trajectories (uses forward kinematic models equations)
  - scores each trajectory and pick the best one

DWB Planner



Global and Local Plans



BT Navigation Server – Default Behavior

- alternates between *navigation* and *recovery*
- navigation
  - compute global path and updated it (replan)
  - pass global path to the controller server that tracks one part of it
  - if either server fails, switch to *contextual recovery* (check if goal changed and clear maps). If this step fails, switch to recovery.
- recovery: cycle through; when one succeeds go back to navigation
  - clear costmaps
  - spin
  - wait
  - backup