Note: This tutorial assumes familiarity with ROS and how to use it. Please see ROS Documentation (/ROS). If you want concise practical example of navigation on simulated robot, this tutorial (http://www.moorerobots.com/blog) provide excellent source.

Please ask about problems and questions regarding this tutorial on answers.ros.org (http://answers.ros.org). Don't forget to include in your question the link to this page, the versions of your OS & ROS, and also add appropriate tags.

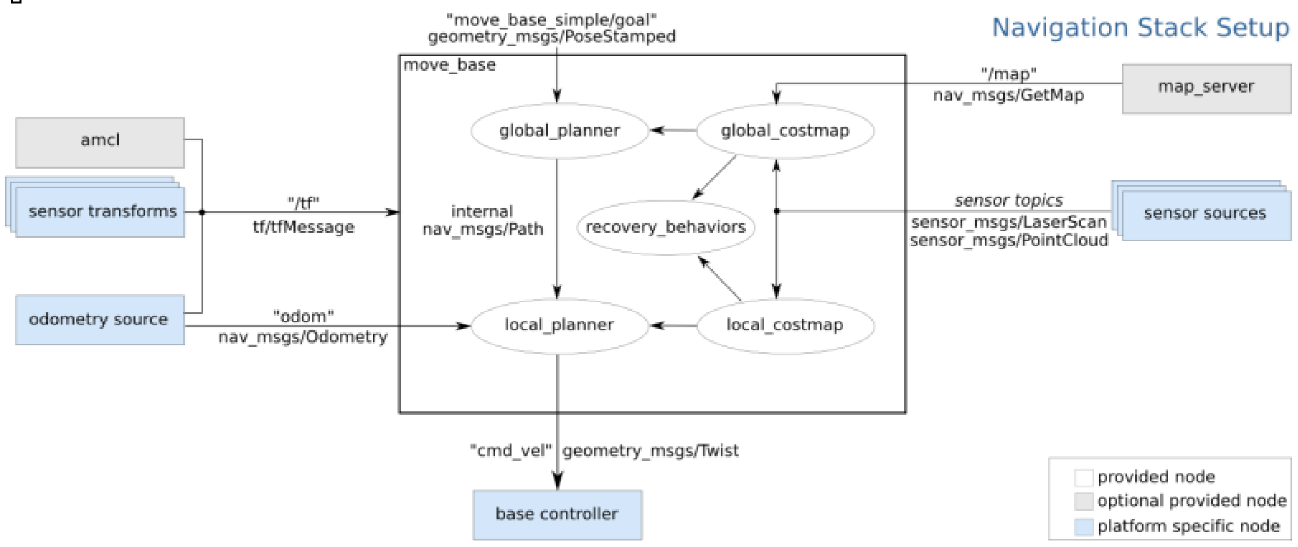
Setup and Configuration of the Navigation Stack on a Robot

Description: This tutorial provides step-by-step instructions for how to get the navigation stack running on a robot. Topics covered include: sending transforms using tf, publishing odometry information, publishing sensor data from a laser over ROS, and basic navigation stack configuration.

Tutorial Level: INTERMEDIATE

Inhaltsverzeichnis

1. Robot Setup



(/navigation/Tutorials/RobotSetup?action=AttachFile&do=view&target=overview_tf.png) The navigation stack assumes that the robot is configured in a particular manner in order to run. The diagram above shows an overview of this configuration. The white components are required components that are already implemented, the gray components are optional components that are already implemented, and the blue components must be created for each robot platform. The pre-requisites of the navigation stack, along with instructions on how to fulfil each requirement, are provided in the sections below.

1.1 ROS

The navigation stack assumes that the robot is using ROS (/ROS). Please consult the ROS documentation (/ROS) for instructions on how to install ROS on your robot.

1.2 Transform Configuration (other transforms)

The navigation stack requires that the robot be publishing information about the relationships between coordinate frames using tf (/tf). A detailed tutorial on setting up this configuration can be found here: Transform Configuration (/navigation/Tutorials/RobotSetup/TF).

1.3 Sensor Information (sensor sources)

The navigation stack uses information from sensors to avoid obstacles in the world, it assumes that these sensors are publishing either `sensor_msgs/LaserScan` or `sensor_msgs/PointCloud` messages over ROS. For information on publishing these messages over ROS, please see the Publishing Sensor Streams Over ROS (/navigation/Tutorials/RobotSetup/Sensors) tutorial. Also, there are a number of sensors that have ROS drivers that already take care of this step. Supported sensors and links to their appropriate drivers are listed below:

- SCIP2.2-compliant Hokuyo Laser Devices as well as the Hokuyo Model 04LX, 30LX - urg_node (/urg_node)
- SICK LMS2xx Lasers - sicktoolbox_wrapper (/sicktoolbox_wrapper)

1.4 Odometry Information (odometry source)

The navigation stack requires that odometry information be published using tf (/tf) and the `nav_msgs/Odometry` message. A tutorial on publishing odometry information can be found here: Publishing Odometry Information Over ROS (/navigation/Tutorials/RobotSetup/Odom). Supported platforms for odometry and links to their appropriate drivers are listed below:

- Videre Erratic: erratic_player (/erratic_player)
- PR2: pr2_mechanism_controllers (/pr2_mechanism_controllers)

1.5 Base Controller (base controller)

The navigation stack assumes that it can send velocity commands using a `geometry_msgs/Twist` message assumed to be in the base coordinate frame of the robot on the "cmd_vel" topic. This means there must be a node subscribing to the "cmd_vel" topic that is capable of taking (vx, vy, vtheta) <==> (cmd_vel.linear.x, cmd_vel.linear.y, cmd_vel.angular.z) velocities and converting them into motor commands to send to a mobile base. Supported platforms for base control and links to their appropriate drivers are listed below:

- Videre Erratic: erratic_player (/erratic_player)
- PR2: pr2_mechanism_controllers (/pr2_mechanism_controllers)

1.6 Mapping (map_server)

The navigation stack does not require a map to operate, but for the purposes of this tutorial, we'll assume you have one. Please see the building a map (/slam_gmapping/Tutorials/MappingFromLoggedData) tutorial for details on creating a map of your environment.

2. Navigation Stack Setup

This section describes how to setup and configure the navigation stack on a robot. It assumes that all the requirements above for robot setup have been satisfied. Specifically, this means that the robot must be publishing coordinate frame information using tf (/tf), receiving `sensor_msgs/LaserScan` or `sensor_msgs/PointCloud` messages from all sensors that are to be used with the navigation stack, and publishing odometry information using both tf (/tf) and the `nav_msgs/Odometry` message while also taking in velocity commands to send to the base. If any of these requirements are not met on your robot, please see the Robot Setup (/navigation/Tutorials/RobotSetup#Robot_Setup) section above for instructions on completing them.

2.1 Creating a Package

This first step for this tutorial is to create a package where we'll store all the configuration and launch files for the navigation stack. This package will have dependencies on any packages used to fulfill the requirements in the Robot Setup (/navigation/Tutorials/RobotSetup#Robot_Setup) section above as well as on the `move_base` package which contains the high-level interface to the navigation stack. So, pick a location for your package and run the following command:

```
catkin_create_pkg my_robot_name_2dnav move_base my_tf_configuration_dep my_odom_configuration_dep my_sensor_configuration_dep
```

This command will create a package with the necessary dependencies to run the navigation stack on your robot.

2.2 Creating a Robot Configuration Launch File

Now that we have a workspace for all of our configuration and launch files, we'll create a roslaunch file that brings up all the hardware and transform publishes that the robot needs. Fire up your favorite editor, and paste the following snippet into a file called `my_robot_configuration.launch`. You should, of course, feel free to replace the text "my_robot" with the name of your actual robot. We'll also have to make similar changes to the launch file as discussed below, so make sure that you read the rest of this section.

```
<launch>

   <node pkg="sensor_node_pkg" type="sensor_node_type" name="sensor_node_name" output="screen">
    <param name="sensor_param" value="param_value" />
 </node>
 <node pkg="odom_node_pkg" type="odom_node_type" name="odom_node" output="screen">
    <param name="odom_param" value="param_value" />
 </node>
 <node pkg="transform_configuration_pkg" type="transform_configuration_type" name="transform_configuration_name" output="screen">
    <param name="transform_configuration_param" value="param_value" />
 </node>

</launch>
```

Ok.. so now we have a template for a launch file, but we need to fill it in for our specific robot. We'll walk through the changes that need to be made in each section below.

```
<launch>

   <node pkg="sensor_node_pkg" type="sensor_node_type" name="sensor_node_name" output="screen">
```

In this section, we'll bring up any sensors that the robot will use for navigation. Replace "sensor_node_pkg" with the name of the package for the ROS driver for your sensor, "sensor_node_type" with the type of the driver for your sensor, "sensor_node_name" with the desired name for your sensor node, and "sensor_param" with any parameters that your node might take. Note that if you have multiple sensors that you intend to use to send information to the navigation stack, you should launch all of them here.

```
 </node>
 <node pkg="odom_node_pkg" type="odom_node_type" name="odom_node" output="screen">
    <param name="odom_param" value="param_value" />
 </node>
```

In this section, we'll launch the odometry for the base. Once again, you'll need to replace the pkg, type, name, and param specifications with those relevant to the node that you're actually launching.

```
    <param name="transform_configuration_param" value="param_value" />
  </node>
```

In this section, we'll launch the transform configuration for the robot. Once again, you'll need to replace the pkg, type, name, and param specifications with those relevant to the node that you're actually launching.

2.3 Costmap Configuration (local_costmap) & (global_costmap)

The navigation stack uses two costmaps to store information about obstacles in the world. One costmap is used for global planning, meaning creating long-term plans over the entire environment, and the other is used for local planning and obstacle avoidance. There are some configuration options that we'd like both costmaps to follow, and some that we'd like to set on each map individually. Therefore, there are three sections below for costmap configuration: common configuration options, global configuration options, and local configuration options.

Note: The following sections cover only basic configuration options for the costmap. For documentation on the full range of options, please see the costmap_2d documentation (/costmap_2d).

2.3.1 Common Configuration (local_costmap) & (global_costmap)

The navigation stack uses costmaps to store information about obstacles in the world. In order to do this properly, we'll need to point the costmaps at the sensor topics they should listen to for updates. Let's create a file called costmap_common_params.yaml as shown below and fill it in:

```
obstacle_range: 2.5
raytrace_range: 3.0
footprint: [[x0, y0], [x1, y1], ... [xn, yn]]
#robot_radius: ir_of_robot
inflation_radius: 0.55

observation_sources: laser_scan_sensor point_cloud_sensor

laser_scan_sensor: {sensor_frame: frame_name, data_type: LaserScan, topic: top
ic_name, marking: true, clearing: true}

point_cloud_sensor: {sensor_frame: frame_name, data_type: PointCloud, topic: t
opic_name, marking: true, clearing: true}
```

Ok, let's break down the file above into manageable parts.

```
obstacle_range: 2.5
raytrace_range: 3.0
```

These parameters set thresholds on obstacle information put into the costmap. The "obstacle_range" parameter determines the maximum range sensor reading that will result in an obstacle being put into the costmap. Here, we have it set at 2.5 meters, which means that the robot will only update its map with information about obstacles that are within 2.5 meters of the base. The "raytrace_range" parameter determines the range to which we will raytrace freespace given a sensor reading. Setting it to 3.0 meters as we have above means that the robot will attempt to clear out space in front of it up to 3.0 meters away given a sensor reading.

```
footprint: [[x0, y0], [x1, y1], ... [xn, yn]]
#robot_radius: ir_of_robot
inflation_radius: 0.55
```

Here we set either the footprint of the robot or the radius of the robot if it is circular. In the case of specifying the footprint, the center of the robot is assumed to be at (0.0, 0.0) and both clockwise and counterclockwise specifications are supported. We'll also set the inflation radius for the costmap. The inflation radius should be set to the maximum distance from obstacles at which a cost should be incurred. For example, setting the inflation radius at 0.55 meters means that the robot will treat all paths that stay 0.55 meters or more away from obstacles as having equal obstacle cost.

```
observation_sources: laser_scan_sensor point_cloud_sensor
```

The "observation_sources" parameter defines a list of sensors that are going to be passing information to the costmap separated by spaces. Each sensor is defined in the next lines.

```
laser_scan_sensor: {sensor_frame: frame_name, data_type: LaserScan, topic: top
ic_name, marking: true, clearing: true}
```

This line sets parameters on a sensor mentioned in `observation_sources`, and this example defines `laser_scan_sensor` as an example. The "frame_name" parameter should be set to the name of the coordinate frame of the sensor, the "data_type" parameter should be set to LaserScan or PointCloud depending on which message the topic uses, and the "topic_name" should be set to the name of the topic that the sensor publishes data on. The "marking" and "clearing" parameters determine whether the sensor will be used to add obstacle information to the costmap, clear obstacle information from the costmap, or do both.

2.3.2 Global Configuration (global_costmap)

We'll create a file below that will store configuration options specific to the global costmap. Open up an editor with the file `global_costmap_params.yaml` and paste in the following text:

```
global_costmap:
  global_frame: /map
  robot_base_frame: base_link
  update_frequency: 5.0
  static_map: true
```

The "global_frame" parameter defines what coordinate frame the costmap should run in, in this case, we'll choose the /map frame. The "robot_base_frame" parameter defines the coordinate frame the costmap should reference for the base of the robot. The "update_frequency" parameter determines the frequency, in Hz, at which the costmap will run its update loop. The "static_map" parameter determines whether or not the costmap should initialize itself based on a map served by the map_server (/map_server). If you aren't using an existing map or map server, set the static_map parameter to `false`.

2.3.3 Local Configuration (local_costmap)

We'll create a file below that will store configuration options specific to the local costmap. Open up an editor with the file `local_costmap_params.yaml` and paste in the following text:

```
local_costmap:
  global_frame: odom
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 2.0
  static_map: false
  rolling_window: true
  width: 6.0
  height: 6.0
  resolution: 0.05
```

The "global_frame", "robot_base_frame", "update_frequency", and "static_map" parameters are the same as described in the Global Configuration (/navigation/Tutorials/RobotSetup#Global_Configuration) section above. The "publish_frequency" parameter determines the rate, in Hz, at which the costmap will publish visualization information. Setting the "rolling_window" parameter to true means that the costmap will remain centered around the robot as the robot moves through the world. The "width," "height," and "resolution" parameters set the width (meters), height (meters), and resolution (meters/cell) of the costmap. Note that its fine for the resolution of this grid to be different than the resolution of your static map, but most of the time we tend to set them equally.

2.3.4 Full Configuration Options

This minimum configuration should get things up and running, but for more details on the configuration options available for the costmap please see the costmap_2d documentation (/costmap_2d).

2.4 Base Local Planner Configuration

The base_local_planner (/base_local_planner) is responsible for computing velocity commands to send to the mobile base of the robot given a high-level plan. We'll need to set some configuration options based on the specs of our robot to get things up and running. Open up a file called `base_local_planner_params.yaml` and paste the following text into it:

Note: This section covers only basic configuration options for the TrajectoryPlanner. For documentation on the full range of options, please see the base_local_planner documentation (/base_local_planner).

```
TrajectoryPlannerROS:
  max_vel_x: 0.45
  min_vel_x: 0.1
  max_vel_theta: 1.0
  min_in_place_vel_theta: 0.4

  acc_lim_theta: 3.2
  acc_lim_x: 2.5
  acc_lim_y: 2.5

  holonomic_robot: true
```

The first section of parameters above define the velocity limits of the robot. The second section defines the acceleration limits of the robot.

2.5 Creating a Launch File for the Navigation Stack

Now that we've got all of our configuration files in place, we'll need to bring everything together into a launch file for the navigation stack. Open up an editor with the file `move_base.launch` and paste the following text into it:

```
<launch>

   <master auto="start"/>
 <!-- Run the map server -->
    <node name="map_server" pkg="map_server" type="map_server" args="$(find my
_map_package)/my_map.pgm my_map_resolution"/>

 <!--- Run AMCL -->
    <include file="$(find amcl)/examples/amcl_omni.launch" />

   <node pkg="move_base" type="move_base" respawn="false" name="move_base" out
put="screen">
    <rosparam file="$(find my_robot_name_2dnav)/costmap_common_params.yaml" co
mmand="load" ns="global_costmap" />
    <rosparam file="$(find my_robot_name_2dnav)/costmap_common_params.yaml" co
mmand="load" ns="local_costmap" />
    <rosparam file="$(find my_robot_name_2dnav)/local_costmap_params.yaml" com
mand="load" />
    <rosparam file="$(find my_robot_name_2dnav)/global_costmap_params.yaml" co
mmand="load" />
    <rosparam file="$(find my_robot_name_2dnav)/base_local_planner_params.yam
l" command="load" />
 </node>

</launch>
```

The only changes that you should need to make to this file are to change the map server to point to a map you've created and to change "amcl_omni.launch" to "amcl_diff.launch" if you have a differential drive robot. For a tutorial on creating a map, please see the building a map (/slam_gmapping/Tutorials/MappingFromLoggedData).

2.6 AMCL Configuration (amcl)

AMCL has many configuration options that will affect the performance of localization. For more information on AMCL please see amcl documentation (/amcl).

3. Running the Navigation Stack

Now that we've got everything set up, we can run the navigation stack. To do this we'll need two terminals on the robot. In one terminal, we'll launch the `my_robot_configuration.launch` file and in the other we'll launch the `move_base.launch` file that we just created.

Terminal 1:

```
roslaunch my_robot_configuration.launch
```

Terminal 2:

```
roslaunch move_base.launch
```

Congratulations, the navigation stack should now be running. For information on sending goals to the navigation stack through a graphical interface, please see the rviz and navigation tutorial (/navigation/Tutorials/Using%20rviz%20with%20the%20Navigation%20Stack). If you want to send goals to the navigation stack using code instead, please see the Sending Simple Navigation Goals (/navigation/Tutorials/SendingSimpleGoals) tutorial.

4. Troubleshooting

For common issues encountered when running the navigation stack, please see the navigation stack troubleshooting (/navigation/Troubleshooting) page.

#keywords mobile platform setup, robot setup, setup robot, getting started with mobile robot