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.

Roslaunch tips for large projects

Description: This tutorial describes some tips for writing roslaunch (/roslaunch) files for large projects. The focus is on how to structure launch files so they may be reused as much as possible in different situations. We'll use the 2dnav_pr2 (/2dnav_pr2) package as a case study.

Tutorial Level: INTERMEDIATE

Next Tutorial: Roslaunch Nodes in Valgrind or GDB (/roslaunch/Tutorials/Roslaunch%20Nodes%20in%20Valgrind%20or%20GDB)

Sommaire

1. Introduction

Large applications on a robot typically involve several interconnected nodes (/Nodes), each of which have many parameters (/Parameter%20Server). 2d navigation is a good example. The 2dnav_pr2 (/2dnav_pr2) application consists of the move_base node itself, localization, ground plane filtering, the base controller, and the map server. Collectively, there are also a few hundred ROS parameters that affect the behavior of these nodes. Finally, there are constraints such as the fact that ground plane filtering should run on the same machine as the tilt laser for efficiency.

A roslaunch file allows us to say all this. Given a running robot, launching the file *2dnav_pr2.launch* in the *2dnav_pr2* package will bring up everything required for the robot to navigate. In this tutorial, we'll go over this launch file and the various features used.

We'd also like roslaunch files to be as reusable as possible. In this case, moving between physically identical robots can be done without changing the launch files at all. Even a change such as moving from the robot to a simulator can be done with only a few changes. We'll go over how the launch file is structured to make this possible.

2. Top-level organization

Here is the top-level launch file (in "`rospack find 2dnav_pr2`/move_base/2dnav_pr2.launch").

```
<launch>
  <group name="wg">
    <include file="$(find pr2_alpha)/$(env ROBOT).machine" />
    <include file="$(find 2dnav_pr2)/config/new_amcl_node.xml" />
    <include file="$(find 2dnav_pr2)/config/base_odom_teleop.xml" />
    <include file="$(find 2dnav_pr2)/config/lasers_and_filters.xml" />
    <include file="$(find 2dnav_pr2)/config/map_server.xml" />
    <include file="$(find 2dnav_pr2)/config/ground_plane.xml" />

    <!-- The navigation stack and associated parameters -->
    <include file="$(find 2dnav_pr2)/move_base/move_base.xml" />
  </group>
</launch>
```

This file includes a set of other files. Each of these included files contains nodes and parameters (and possibly nested includes) pertaining to one part of the system, such as localization, sensor processing, and path planning.

Design tip: Top-level launch files should be short, and consist of include's to other files corresponding to subcomponents of the application, and commonly changed ROS parameters.

This makes it easy to swap out one piece of the system, as we'll see later.

To run this on the PR2 robot requires bringing up a core (/roscore), then bringing up a robot-specific launch file such as *pre.launch* in the *pr2_alpha* package, and then launching *2dnav_pr2.launch*. We could have included a robot launch file here rather than requiring it to be launched separately. That would bring the following tradeoffs:

- PRO: We'd have to do one fewer "open new terminal, roslaunch" step.
- CON: Launching the robot launch file initiates a calibration phase lasting about a minute long. If the 2dnav_pr2 launch file included the robot launch file, every time we killed the roslaunch (with control-c) and brought it back up, the calibration would happen again.
- CON: Some of the 2d navigation nodes require that the calibration already have finished before they start. Roslaunch intentionally does not provide any control on the order or timing of node start up. The ideal solution would be to make nodes work gracefully by waiting till calibration is done, but pending that, putting things in two launch files allows us to launch the robot, wait until calibration is complete, then launch 2dnav.

There is therefore no universal answer on whether or not to split things into multiple launch files. Here, it has been decided to use two different launch files.

Design tip: Be aware of the tradeoffs when deciding how many top-level launch files your application requires.

3. Machine tags and Environment Variables

We would like control over which nodes run on which machines, for load-balancing and bandwidth management. For example, we'd like the amcl (/amcl) node to run on the same machine as the base laser. At the same time, for reusability, we don't want to hardcode machine names into roslaunch files. Roslaunch handles this with **machine tags**.

The first include is

```
<include file="$(find pr2_alpha)/$(env ROBOT).machine" />
```

The first thing to note about this file is the use of the *env* substitution argument to use the value of the environment variable ROBOT. For example, doing

```
export ROBOT=pre
```

prior to the roslaunch would cause the file *pre.machine* to be included.

Design tip: Use the env substitution argument to allow parts of a launch file to depend on environment variables.

Next, let's look at an example machine file: *pre.machine* in the *pr2_alpha* package.

```
<launch>
  <machine name="c1" address="pre1" ros-root="$(env ROS_ROOT)" ros-package-pat
h="$(env ROS_PACKAGE_PATH)" default="true" />
  <machine name="c2" address="pre2" ros-root="$(env ROS_ROOT)" ros-package-pat
h="$(env ROS_PACKAGE_PATH)" />
</launch>
```

This file sets up a mapping between logical machine names, "c1" and "c2" in this case, and actual host names, such as "pre2". It even allows controlling the user you log in as (assuming you have the appropriate ssh credentials).

Once the mapping has been defined, it can be used when launching nodes. For example, the included file *config/new_amcl_node.xml* in the *2dnav_pr2* package contains the line

```
<node pkg="amcl" type="amcl" name="amcl" machine="c1">
```

This causes the *amcl* node to run on machine with logical name *c1* (looking at the other launch files, you'll see that most of the laser sensor processing has been put on this machine).

When running on a new robot, say one known as *prf*, we just have to change the ROBOT environment variable. The corresponding machine file (*prf.machine* in the *pr2_alpha* package) will then be loaded. We can even use this for running on a simulator, by setting ROBOT to *sim*. Looking at the file *sim.machine* in the *pr2_alpha* package, we see that it just maps all logical machine names to localhost.

Design tip: Use machine tags to balance load and control which nodes run on the same machine, and consider having the machine file name depend on an environment variable for reusability.

4. Parameters, namespaces, and yaml files

Let's look at the included file *move_base.xml*. Here is a portion of this file:

```
<node pkg="move_base" type="move_base" name="move_base" machine="c2">
  <remap from="odom" to="pr2_base_odometry/odom" />
  <param name="controller_frequency" value="10.0" />
  <param name="footprint_padding" value="0.015" />
  <param name="controller_patience" value="15.0" />
  <param name="clearing_radius" value="0.59" />
  <rosparam file="$(find 2dnav_pr2)/config/costmap_common_params.yaml" comman
d="load" ns="global_costmap" />
  <rosparam file="$(find 2dnav_pr2)/config/costmap_common_params.yaml" comman
d="load" ns="local_costmap" />
  <rosparam file="$(find 2dnav_pr2)/move_base/local_costmap_params.yaml" comma
nd="load" />
  <rosparam file="$(find 2dnav_pr2)/move_base/global_costmap_params.yaml" comm
and="load" />
  <rosparam file="$(find 2dnav_pr2)/move_base/navfn_params.yaml" command="loa
d" />
  <rosparam file="$(find 2dnav_pr2)/move_base/base_local_planner_params.yaml"
command="load" />
</node>
```

This fragment launches the *move_base* node. The first included element is a **remapping**. Move_base is designed to receive odometry on the topic "odom". In the case of the pr2, odometry is published on the *pr2_base_odometry* topic, so we remap it.

Design tip: Use topic remapping when a given type of information is published on different topics in different situations.

This is followed by a bunch of <param> tags. Note that these parameters are inside the node element (since they're before the </node> at the end), so they will be private parameters (/Parameter%20Server#Private_Parameters). For example, the first one sets *move_base/controller_frequency* to 10.0.

After the <param> elements, there are some <rosparam> elements. These read parameter data in yaml, a format which is human readable and allows complex data structures. Here's a portion of the *costmap_common_params.yaml* file loaded by the first <rosparam> element:

```
raytrace_range: 3.0
footprint: [[-0.325, -0.325], [-0.325, 0.325], [0.325, 0.325], [0.46, 0.0],
[0.325, -0.325]]
inflation_radius: 0.55

# BEGIN VOXEL STUFF
observation_sources: base_scan_marking base_scan tilt_scan ground_object_cloud

base_scan_marking: {sensor_frame: base_laser, topic: /base_scan_marking, data_
type: PointCloud, expected_update_rate: 0.2,
  observation_persistence: 0.0, marking: true, clearing: false, min_obstacle_h
eight: 0.08, max_obstacle_height: 2.0}
```

We see that yaml allows things like vectors (for the *footprint* parameter). It also allows putting some parameters into a nested namespace (/Names). For example, *base_scan_marking/sensor_frame* is set

to *base_laser*. Note that these namespaces are relative to the yaml file's own namespace, which was declared as *global_costmap* by the *ns* attribute of the including *rosparam* element. In turn, since that *rosparam* was included by the node element, the fully qualified name of the parameter is */move_base /global_costmap/base_scan_marking/sensor_frame*.

The next line in move_base.xml is:

```
<rosparam file="$(find 2dnav_pr2)/config/costmap_common_params.yaml" command="
load" ns="local_costmap" />
```

This actually includes the exact same yaml file as the line before it. It's just in a different namespace (the local_costmap namespace is for the trajectory controller, while the global_costmap namespace affects the global navigation planner). This is much nicer than having to retype all the values.

The next line is:

```
<rosparam file="$(find 2dnav_pr2)/move_base/local_costmap_params.yaml" comman
d="load"/>
```

Unlike the previous ones, this element doesn't have an *ns* attribute. Thus the yaml file's namespace is the parent namespace, */move_base*. But take a look at the first few lines of the yaml file itself:

```
local_costmap:
  #Independent settings for the local costmap
  publish_voxel_map: true
  global_frame: odom_combined
  robot_base_frame: base_link
```

Thus we see that the parameters are in the */move_base/local_costmap* namespace after all.

Design tip: Yaml files allow parameters with complex types, nested namespaces of parameters, and reusing the same parameter values in multiple places.

5. Reusing launch files

The motivation for many of the tips above was to make reusing launch files in different situations easier. We've already seen one example, where the use of the *env* substitution arg can allow modifying behavior without changing any launch files. There are some situations, though, where that's inconvenient or impossible. Let's take a look at the pr2_2dnav_gazebo (/pr2_2dnav_gazebo) package. This contains a version of the 2d navigation app, but for use in the Gazebo simulator (/simulator_gazebo). For navigation, the only thing that changes is actually that the Gazebo environment we use is based on a different static map, so the map_server node must be loaded with a different argument. We could have used another env substitution here. But that would require the user to set a bunch of environment variables just to be able to roslaunch. Instead, 2dnav gazebo contains its own top level launch file called '2dnav-stack-amcl.launch', shown here (modified slightly for clarity):

```
<launch>
  <include file="$(find pr2_alpha)/sim.machine" />
  <include file="$(find 2dnav_pr2)/config/new_amcl_node.xml" />
  <include file="$(find 2dnav_pr2)/config/base_odom_teleop.xml" />
  <include file="$(find 2dnav_pr2)/config/lasers_and_filters.xml" />
  <node name="map_server" pkg="map_server" type="map_server" args="$(find gaze
bo_worlds)/Media/materials/textures/map3.png 0.1" respawn="true" machine="c1"
/>
  <include file="$(find 2dnav_pr2)/config/ground_plane.xml" />
  <!-- The naviagtion stack and associated parameters -->
  <include file="$(find 2dnav_pr2)/move_base/move_base.xml" />
</launch>
```

The first difference is that, since we know we're on the simulator, we just use the sim.machine file rather than using a substitution argument. Second, the line

```
<include file="$(find 2dnav_pr2)/config/map_server.xml" />
```

has been replaced by

```
<node name="map_server" pkg="map_server" type="map_server" args="$(find gazebo
_worlds)/Media/materials/textures/map3.png 0.1" respawn="true" machine="c1" />
```

The included file in the first case just contained a node declaration as in the second case, but with a different map file.

Design tip: To modify a "top-level" aspect of an application, copy the top level launch file and change the portions you need.

6. Parameter overrides

The technique above sometimes becomes inconvenient. Suppose we want to use *2dnav_pr2*, but just change the resolution parameter of the local costmap to 0.5. We could just locally change *local_costmap_params.yaml*. This is the simplest for temporary modifications, but it means we can't check the modified file back in. We could instead make a copy of *local_costmap_params.yaml* and modify it. We would then have to change *move_base.xml* to include the modified yaml file. And then we would have to change *2dnav_pr2.launch* to include the modified *move_base.xml*. This can be time-consuming, and if using version control, we would no longer see changes to the original files. An alternative is to restructure the launch files so that the *move_base/local_costmap/resolution* parameter is defined in the top-level file *2dnav_pr2.launch*, and make a modified version of just that file. This is a good option if we know in advance which parameters are likely to be changed.

Another option is to use roslaunch's overriding behavior: parameters are set in order (after includes are processed). Thus, we could make a further top-level file that overrides the original resolution:

```
<launch>
<include file="$(find 2dnav_pr2)/move_base/2dnav_pr2.launch" />
<param name="move_base/local_costmap/resolution" value="0.5"/>
</launch>
```

The main drawback is that this method can make things harder to understand: knowing the actual value that roslaunch sets for a parameter requires tracing through the including roslaunch files. But it does avoid having to make copies of multiple files.

Design tip: To modify a deeply nested parameter in a tree of launch files which you cannot change, use roslaunch's parameter overriding semantics.

7. Roslaunch arguments

As of CTurtle, roslaunch has an argument substitution feature together with tags that allow conditioning parts of the launch file on the value of arguments. This can be a more general and clear way to structure things than the parameter override mechanism or launch file reuse techniques above, at the cost of having to modify the original launch file to specify what the changeable arguments are. See the roslaunch XML documentation (/roslaunch/XML).

Design tip: If you can modify the original launch file, it's often preferable to use roslaunch arguments rather than parameter overriding or copying roslaunch files.

Wiki: ROS/Tutorials/Roslaunch tips for larger projects (dernière édition le 2012-11-16 19:06:20 par WilliamWoodall (/WilliamWoodall))