5

Working with Pluginlib, Nodelets and Gazebo Plugins

★ Understanding pluginlib

⇒ Plugins are modular piece of software which can add a new feature to the existing ~~[illegible]~~ software application.

↳ Using this method, we can extend the capabilities of software to any level.

⇒ When we are going to build a complex ROS based application for a robot, plugins will be a good choice to extend the capabilities of the application.

⇒ The ROS system provides a plugin framework called pluginlib to dynamically load / unload plugins, which can be a library or class.

★ Creating plugins for the calculator application using pluginlib

⇒ The aim of this example is to show how to add new feature to calculator without modifying main application code.

{ Catkin_Create_pkg pluginlib_calculator
pluginlib roscpp std_msgs }

Step 1: Creating calculator-base header file

=> main purpose of this file is to declare functions/methods that are commonly used by the plugins.

namespace calculator_base

└> Class => calc_functions

  └> Member functions

    └> get_numbers

    └> operation

Step2: Creating calculator_plugins header file

=> Main functions of this * calculator Plugins, which are named as Add, Sub, Mul, and Div.

{ file is to define complete function of the }

Step3 - Exporting plugins using calculator -plugins.cpp

⇒ In order to load the class of plugins dynamically, we have to export each class using a special macro called PLUGINLIB_EXPORT_CLASS.

```
#include <pluginlib/class_list_macros.h>

PLUGINLIB_EXPORT_CLASS (calculator
   -plugins:: Add, calculator_base ::
      calc_function);
```

Step4 - Implementing plugin loader using calculator -loader.cpp

⇒ The plugin loader node loads each plugin and inputs the number to each plugin and fetch's the result from the plugin.

```
# include <boost/shared_ptr.hpp>
# include <pluginlib/class_loader.h>

Pluginlib:: ClassLoader <calculator
-base:: calc_functions>
calc_loader ("pluginlib_calculator"
, "calculator_base:: calc_functions");
```

```
boost:: shared_ptr <Calculator_base ::
calc_functions> add = Calc_loader.
CreateInstance ("pluginlib_Calculator/Add");
```

Step 5: Creating plugin description file : Calculator_plugin.xml

```
<library path="lib/libpluginlib_Calculator">

<Class name="pluginlib_Calculator/Add"
       type="Calculator_plugins :: Add"
       base_class_type="Calculator_base:: calc_function">

  <description>This is add plugin.</description>

</Class>
    .
    .
    .
    .
    .
</library>
```

Step 6: Registering plugin with the ROS Package system

⇒ For pluginlib to fine all plugins based Packages in the ROS System, we should export the plugin description file inside Package.xml.

=> If we do not include this plugin, the ROS system wont find the plugins inside the package.

```
<export>
<pluginlib_Calculator plugin="${Prefix}
 /calculator_plugins.xml" />
</export>

<build_depend> pluginlib_calculator
            </build_depend>
<run_depend> pluginlib_calculator
                   </run_depend>
```

Step 7 - Editing the CMakeList.txt file

```
add_library (pluginlib_calculator
              src/calculator_plugins.cpp)
target_link_library (pluginlib_calculator
              ${Catkin_LIBRARIES})

add_executable (Calculator_loader
              src/calculator_loader
              .cpp)

target_link_libraries (Calculator_loader ${Catkin
                                 _LIBRARIES})
```

Step 8: Querying the list of plugins in a Package

```
rospak plugins --attrib=plugin pluginlib_calculator
```

Step 9: Running the plugin loader

```
rosrun pluginlib_calculator calculator_loader
```

★ Understanding ROS nodelets

⇒ Nodelets are a type of ROS node that are designed to run multiple nodes in a Single process, with each node running as a thread.

↳ These threaded nodes can communicate with external nodes too.

⇒ In nodelets, we dynamically load each class as a plugin, which has a Seperate namespace.

⇒ Nodelets are used when the Volume of data transferred between nodes are very high.

<u>example</u> ⇒ 3D Sensor

* Creating a Nodelet

"We are going to Create a basic nodelet that Can Subscribe a String topil called /msg-in and Publishes the Same String on the topic /msg_out"

Step1- Creating a package for nodelet

Step2- Creating hell_world.cpp nodelet

⇒ we should include class_list_macro.h and nodelet.h to access pluginlib API's and Nodelets APIs.

```
namespace nodelet_hello_world
 {
  Class Hello : public nodelet::Nodelet
   {
```

⇒ All nodelet Class should inherit from the nodelet base class and be dynamically loadable using pluginlib.

```
Virtual void onInit()
 {
  ros::NodeHandle& private_nh = getPrivateN
                                odeHandle();
NODELET_DEBUG ("Initialized the Nodelet")
Pub = private_nh.advertise <std_msgs:: String>
                          ("msg-out", 5);

sub= private_nh.subscribe("msg-in", 5, &Hello::
                          callback, this);

}
```

=> This is the initilization function of a nodelet.

↳ Inside this function we are creating a node handle object, topic publisher and Subscriber on the topic msg-out and msg-in respectively.

```
PLUGINLIB_EXPORT_CLASS
  (nodelet_hello_world:: Hello, nodelet:: Nodelet);
```

=> Here we are exporting the Hello as a plugin for the dynamic loading.

Step 4 – Creating plugin description file

hello_world.xml

```
<library path="libnodelet_hello_world">
  <class name ="nodelet_hello_world/
    Hello" type="nodelet_hello_world::HEllo"
    base_class_type="nodelet::Nodelet">
   <description>
    A node to republish a message
   </description>
  </class>
 </library>
```

Step 5 – Adding the export tag in Pakage.xml

We need to add the export tag in Package.xml and also add build and run dependencies.

Step 6 – Editing CMakeList.txt

```
add_libraries(nodelet_hello_world
              src/hello_world.cpp)

target_link_libraries(nodelet_hello_world
                      ${catkin_LIBRARIES}
                      )
```

Step7 - Building and running nodelets

=> The first step in running nodelets is to start the nodelet manager.

It is a C++ executable program, which will listen to the ROS Service and dynamically load nodelets

=> To start nodelet manager:

```
rosrun nodelet nodelet manager __name:=
                    nodelet_Manager
```

=> After launching the nodelet manager, we can start the nodelet by using the following commands:

```
rosrun nodelat nodelet load nodelet_hello
            _world/Hello node_manager
            __Name:=nodelet1
```

=> Creates instance of nodelet_hello_world /Hello nodelet with a name nodelet1.

Step 8 – Creating launch files for nodelets

```
<launch>

<node pkg="nodelet" type="nodelet"
 name="standalone_nodelet"
  args="manager" output="screen"/>

 <node pkg="nodelet" type="nodelet"
  name="test1" args="load nodelet_hello
  -world/Hello standalone_nodelet"
  output="screen"/>

</launch>
```

★ Gazebo Plugins

⇒ Gazebo plugins help us to Control the robot models, sensors, world properties, and even the way Gazebo runs.

⇒ We can Mainly classify the Plugins as follows:

- World plugin

  ↳ using this we can Control the properties of a specific world in Gazebo.

- Model plugin
  ↳ Parameter such as joint state of the model, control of the joints and so, on can be controlled using this plugin.

- Sensor plugin
  ↳ These are for modelling Sensors such as Camera, IMU, and so on in Gazebo.

- System plugin
  ↳ Control System related function in Gazebo.

- Visual plugin
  ↳ Visual properties of any Gazebo component can be accessed and controlled using the visual plugin.

⇒ Gazebo plugins are independent of ROS and we dont need ROS libraries to build the plugin.