

We will get the following result if everything is built properly:

```
jcacace@robot:~$ rospack plugins --attrib=plugin pluginlib_calculator
pluginlib_calculator /home/jcacace/catkin_ws/src/MASTERING_ROS/ch6/pluginlib_calculator/calculator_
plugins.xml
```

Figure 2: The result of the plugin query

Step 9 - Running the plugin loader

After launched the roscore, we can execute the `calculator_loader` using the following command:

```
$ rosrun pluginlib_calculator calculator_loader
```

The following screenshot shows the output of this command, to check whether everything is working fine. The loader gives both inputs as `10.0` and we are getting the proper result, as shown, using plugins in the screenshot:

```
jcacace@robot:~$ rosrun pluginlib_calculator calculator_loader
[ INFO] [1506769896.353657043]: Triangle area: 20.00
[ INFO] [1506769896.353796789]: Substracted result: 0.00
[ INFO] [1506769896.353853201]: Multiplied result: 100.00
[ INFO] [1506769896.353886772]: Division result: 1.00
```

Figure 3: Result of the plugin loader node

In the next section, we will look at a new concept called **nodelets** and discuss how to implement them.

Understanding ROS nodelets

Nodelets are specific ROS nodes designed to run multiple algorithms within the same process in an efficient way, executing each process as threads. The threaded nodes can communicate with each other efficiently without overloading the network, with zero copy transport between two nodes. These threaded nodes can communicate with external nodes too.

As we did using pluginlib, in nodelets, we can also dynamically load each class as a plugin, which has a separate namespace. Each loaded class can act as separate nodes, which are on a single process called nodelet.

Nodelets are used when the volume of data transferred between nodes are very high; for example, in transferring data from 3D sensors or cameras.

Next, we will look at how to create a nodelet.

Creating a nodelet

In this section, we are going to create a basic nodelet that can subscribe a string topic called `/msg_in` and publish the same string (`std_msgs/String`) on the topic `/msg_out`.

Step 1 - Creating a package for a nodelet

We can create a package called `nodelet_hello_world`, using the following command to create our nodelet:

```
$ catkin_create_pkg nodelet_hello_world nodelet roscpp std_msgs
```

Otherwise, we can use the existing package from `nodelet_hello_world`, or download it from the following link:

```
$ git clone https://github.com/jocacace/nodelet_hello_world
```

Here, the main dependency of this package is the `nodelet` package, which provides APIs to build a ROS nodelet.

Step 2 - Creating the `hello_world.cpp` nodelet

Now, we are going to create the nodelet code. Create a folder called `src` inside the package and create a file called `hello_world.cpp`.

You will get the existing code from the `nodelet_hello_world/src` folder.

Step 3 - Explanation of `hello_world.cpp`

Here is the explanation of the code:

```
#include <pluginlib/class_list_macros.h>
#include <nodelet/nodelet.h>
#include <ros/ros.h>
#include <std_msgs/String.h>
#include <stdio.h>
```

These are the header files of this code. We should include `class_list_macro.h` and `nodelet.h` to access the `pluginlib` APIs and `nodelets` APIs:

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

Here, we create a `nodelet` class called `Hello`, which inherits a standard `nodelet` base class. All `nodelet` classes should inherit from the `nodelet` base class and be dynamically loadable using `pluginlib`. Here, the `Hello` class is going to be used for dynamic loading:

```
virtual void onInit()
{
    ros::NodeHandle& private_nh = getPrivateNodeHandle();
    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 initialization function of a `nodelet`. This function should not block or do significant work. Inside the function, we are creating a node handle object, topic publisher, and subscriber on the topic `msg_out` and `msg_in`, respectively. There are macros to print debug messages while executing a `nodelet`. Here, we use `NODELET_DEBUG` to print debug messages in the console. The subscriber is tied up with a callback function called `callback()`, which is inside the `Hello` class:

```
void callback(const std_msgs::StringConstPtr input)
{
    std_msgs::String output;
    output.data = input->data;
    NODELET_DEBUG("Message data = %s", output.data.c_str());
    ROS_INFO("Message data = %s", output.data.c_str());
    pub.publish(output);
}
```

In the `callback()` function, it will print the messages from the `/msg_in` topic and publish to the `/msg_out` topic:

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

Here, we are exporting `Hello` as a plugin for the dynamic loading.

Step 4 - Creating the plugin description file

Like the `pluginlib` example, we have to create a plugin description file inside the `nodelet_hello_world` package. The plugin description file, `hello_world.xml`, is as follows:

```
<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 package.xml

We need to add the export tag in `package.xml` and add build and run dependencies:

```
<export>
  <nodelet plugin="${prefix}/hello_world.xml"/>
</export>

<build_depend>nodelet_hello_world</build_depend>
<run_depend>nodelet_hello_world</run_depend>
```

Step 6 - Editing CMakeLists.txt

We need to add additional lines of code in `CMakeLists.txt` to build a nodelet package. Here are the extra lines. You will get the complete `CMakeLists.txt` file from the existing package itself:

```
## Declare a cpp library
add_library(nodelet_hello_world
  src/hello_world.cpp
)

## Specify libraries to link a library or executable target against
target_link_libraries(nodelet_hello_world
  ${catkin_LIBRARIES}
)
```

Step 7 - Building and running nodelets

After following this procedure, we can build the package using `catkin_make` and, if the build is successful, we can generate the shared object `libnodelet_hello_world.so` file, which represents the plugin.

The first step in running nodelets is to start the **nodelet manager**. A nodelet manager is a C++ executable program, which will listen to the ROS services and dynamically load nodelets. We can run a standalone manager or can embed it within a running node.

The following commands can start the nodelet manager:

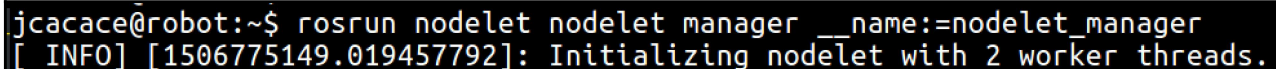
Start `roscore`:

```
$ roscore
```

Start the nodelet manager, using the following command:

```
$ rosrun nodelet nodelet manager __name:=nodelet_manager
```

If the nodelet manager runs successfully, we will get a message, as shown here:



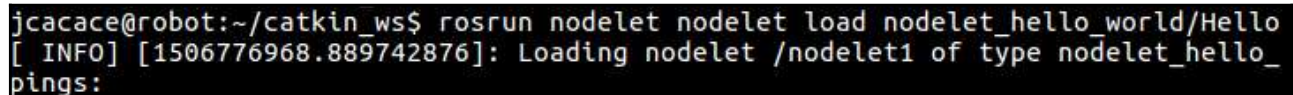
```
jcacace@robot:~$ rosrun nodelet nodelet manager __name:=nodelet_manager
[ INFO] [1506775149.019457792]: Initializing nodelet with 2 worker threads.
```

Figure 4: Running the nodelet manager

After launching the nodelet manager, we can start the nodelet by using the following command:

```
$ rosrun nodelet nodelet load nodelet_hello_world/Hello nodelet_manager
__name:=nodelet1
```

When we execute the preceding command, the nodelet contacts the nodelet manager to instantiate an instance of the `nodelet_hello_world/Hello` nodelet with a name of `nodelet1`. The following screenshot shows the message when we load the nodelet:



```
jcacace@robot:~/catkin_ws$ rosrun nodelet nodelet load nodelet_hello_world/Hello
[ INFO] [1506776968.889742876]: Loading nodelet /nodelet1 of type nodelet_hello_
pings:
```

Figure 5: Running the nodelet

The topics generated after running this nodelet and the list of nodes are shown here:

```
jcacace@robot:~$ rostopic list
/nodelet1/msg_in
/nodelet1/msg_out
/nodelet_manager/bond
/rosout
/rosout_agg
```

Figure 6: The list of topics of the nodelet

We can test the node by publishing a string to the `/nodelet1/msg_in` topic and check whether we receive the same message in `nodelet1/msg_out`.

The following command publishes a string to `/nodelet1/msg_in`:

```
$ rostopic pub /nodelet1/msg_in std_msgs/String "Hello"
```

```
jcacace@robot:~$ rostopic pub /nodelet1/msg_in std_msgs/
String "Hello"
publishing and latching message. Press ctrl-C to termin
ate
jcacace@robot: ~
jcacace@robot:~$ rostopic echo /nodelet1/msg_out
data: Hello
```

Figure 7: Publishing and subscribing using the nodelet

We can echo the `msg_out` topic and can confirm whether the code is working properly.

Here, we have seen that a single instance of the `Hello()` class is created as a node. We can create multiple instances of the `Hello()` class with different node names inside this nodelet.

Step 8 - Creating launch files for nodelets

We can also write launch files to load more than one instance of the nodelet class. The following launch file will load two nodelets, with the names `test1` and `test2`, and we can save it with the name `hello_world.launch`:

```
<launch>

<!-- Started nodelet manager -->

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

<!-- Starting first nodelet -->

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

<!-- Starting second nodelet -->

  <node pkg="nodelet" type="nodelet" name="test2" args="load
nodelet_hello_world/Hello standalone_nodelet" output="screen">
  </node>

</launch>
```

The preceding launch can be launched with the following command:

```
$ roslaunch nodelet_hello_world hello_world.launch
```

The following message will show up on the Terminal if it is launched successfully:

```
[ INFO] [1506951118.603857605]: Loading nodelet /test2 of type nodelet_hello_world/Hello to manager
standalone_nodelet with the following remappings:
[ INFO] [1506951118.606768479]: Loading nodelet /test1 of type nodelet_hello_world/Hello to manager
standalone_nodelet with the following remappings:
[ INFO] [1506951118.610320371]: waitForService: Service [/standalone_nodelet/load_nodelet] has not
been advertised, waiting...
[ INFO] [1506951118.613444334]: waitForService: Service [/standalone_nodelet/load_nodelet] has not
been advertised, waiting...
[ INFO] [1506951118.627001318]: Initializing nodelet with 2 worker threads.
[ INFO] [1506951118.632595864]: waitForService: Service [/standalone_nodelet/load_nodelet] is now a
vailable.
[ INFO] [1506951118.634985422]: waitForService: Service [/standalone_nodelet/load_nodelet] is now a
vailable.
```

Figure 8: Launching multiple instances of the `Hello()` class.

The list of topics and nodes are shown here. We can see two nodelets instantiated and we can see their topics too:

```

jcacace@robot:~$ rostopic list
/rosout
/rosout_agg
/standalone_nodelet/bond
/test1/msg_in
/test1/msg_out
/test2/msg_in
/test2/msg_out
jcacace@robot:~$ rosnode list
/rosout
/standalone_nodelet
/test1
/test2

```

Figure 9: Topics generated by the multiple instances of Hello() class.

Topics are generated by the multiple instances of the `Hello()` class. We can see the interconnection between these nodelets using the `rqt_graph` tool. Open `rqt_gui`:

```
$ rosrun rqt_gui rqt_gui
```

Load the Node Graph plugin from the following option, **Plugins | Introspection | Node Graph**, and you will get the graph as shown in the following figure:

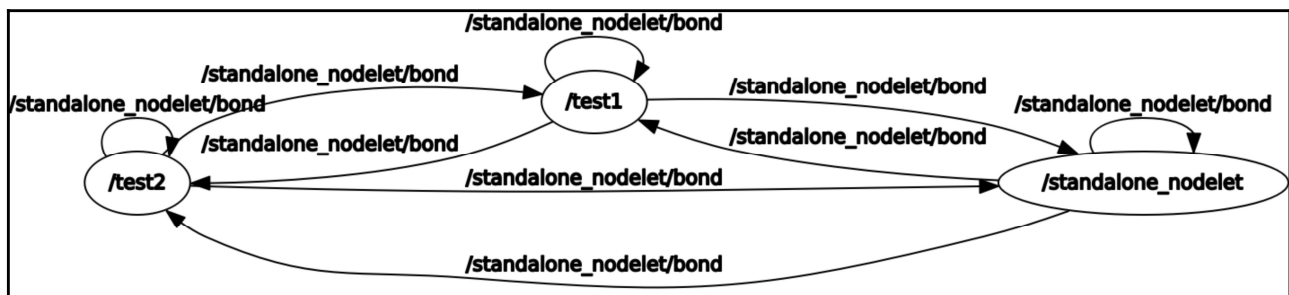


Figure 10: A two-node instance of a nodelet

Alternatively, you can directly load the `rqt_node` plugin:

```
$ rosrun rqt_graph rqt_graph
```