Robot Vision Systems

Lecture 15: ROS Nodes in C++

Michael Felsberg

michael.felsberg@liu.se

Fixes / Correction

- ROS Jade supports OpenCV3, but some parts of documentation are behind
- Use OpenCV 3.0:
  - package.xml:

    ```
    <build_depend>opencv</build_depend>
    ```

  - CMakeLists.txt:

    ```
    find_package(OpenCV REQUIRED)
    include_directories(
           ${OpenCV_INCLUDE_DIRS})
    target_link_libraries(
           ${OpenCV_LIBRARIES})
    ```

Publisher Node in C++

- “Talker”, continually broadcasting a message
- part of your package, e.g.

  ```
  $ roscd beginner_tutorials
  ```

- code is located in ‘src’
- type or download

  ```
  $ wget https://raw.github.com/ros/ros_tutorials/jade-devel/roscpp_tutorials/talker/talker.cpp
  ```

Publisher Node in C++

```
#include "ros/ros.h“                    // meta header
#include "std_msgs/String.h“            // see Python
#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");      // name through cmdline
  ros::NodeHandle n;                    // first handle inits
  ros::Publisher chatter_pub =          // node publishes on topic
            n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(10);              // see Python version
  int count = 0;
  while (ros::ok())                     // Ctrl-C/SIGINT: false
```

Publisher Node in C++

```
  {
    std_msgs::String msg;
    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();              // message type string
    ROS_INFO("%s", msg.data.c_str()); // instead of printf
    chatter_pub.publish(msg);         // broadcast
    ros::spinOnce();                  // good practice (callbacks)
    loop_rate.sleep();                // see Python version
    ++count;
  }
  return 0;
}
```

Subscriber Node in Python

- type or download

```
$ wget https://raw.github.com/ros/ros_tutorials/jade-devel/roscpp_tutorials/listener/listener.cpp
```

Subscriber Node in C++

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs
     ::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]",
     msg->data.c_str());
// message is boost shared_ptr
}
```

Subscriber Node in C++

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;    // see before
  ros::Subscriber sub = n.subscribe(
   "chatter", 1000, chatterCallback);
  ros::spin();
// spins until ros::ok() is false
  return 0;
}
```

Changes in CMakeLists.txt

```
include_directories(include
                 ${catkin_INCLUDE_DIRS})


add_executable(talker src/talker.cpp)
target_link_libraries(talker
                 ${catkin_LIBRARIES})
add_dependencies(talker
beginner_tutorials_generate_messages_cpp)


add_executable(listener src/listener.cpp)
target_link_libraries(listener
                 ${catkin_LIBRARIES})
add_dependencies(listener
beginner_tutorials_generate_messages_cpp)
```

Testing the Nodes

- run in `catkin_ws` (suggestion: add first line to `.bashrc`)

```
$ source ./devel/setup.bash
$ catkin_make
$ roscore
$ rosrun beginner_tutorials
talker
$ rosrun beginner_tutorials
listener
```

Less Trivial Example

- Like for Python: “image hello world” split into two nodes
  - “imtalker” acquiring images
  - “imlistener” showing images
- Working USB-cam required
  - VirtualBox: tick under “Devices/Webcams”
  - you might have to install ros-jade-usb-cam
  - for some reason different from Python: acquisition works in VirtualBox

Image Publisher (C++)

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>

int main(int argc, char **argv) {
  ros::init(argc, argv, "imtalker");
  ros::NodeHandle n;
  image_transport::ImageTransport it(n);  // new
  image_transport::Publisher pub = it.advertise(
                           "camera/image", 1);
  cv::VideoCapture cap(0);
  if(!cap.isOpened()) return 1;
  cv::Mat frame;
```

Image Publisher (C++)

```
  sensor_msgs::ImagePtr msg;   // will be copy
  ros::Rate loop_rate(10);
  while (n.ok()) {
    cap >> frame;
    if(!frame.empty()) {
      msg = cv_bridge::CvImage(std_msgs::Header(
               ), "bgr8", frame).toImageMsg();
      pub.publish(msg);
    }
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

Image Subscriber (C++)

```
#include <ros/ros.h>
#include <image_transport/
                   image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>

void imageCallback(const sensor_msgs::
                   ImageConstPtr& msg) {
  cv::imshow("view", cv_bridge::
      toCvShare(msg, "bgr8")->image);
  cv::waitKey(30);
}
```

Image Subscriber (C++)

```
int main(int argc, char **argv) {
  ros::init(argc, argv, "image_listener");
  ros::NodeHandle n;
  cv::namedWindow("view");
  cv::startWindowThread();      // recommended
  image_transport::ImageTransport it(n);
  image_transport::Subscriber sub = it.
   subscribe("camera/image",1,imageCallback);
  ros::spin();
  cv::destroyWindow("view");
  return 0;
}
```

Services

- Simple example: adding two numbers
- Makes use of `AddTwoInts.srv` (lecture 10)
- Code placed in `src/`
- Two scripts:
  - server `add_two_ints_server.py`
  - client `add_two_ints_client.py`

Server in C++

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"

bool add(beginner_tutorials::AddTwoInts::
    Request &req, beginner_tutorials::
    AddTwoInts::Response &res) {
  res.sum = req.a + req.b;
  ROS_INFO("request: x=%ld, y=%ld", (long
                   int)req.a, (long int)req.b);
  ROS_INFO("sending back response: [%ld]",
                    (long int)res.sum);
  return true;
}
```

Server in C++

```
int main(int argc, char **argv)
{
  ros::init(argc, argv,
               "add_two_ints_server");
  ros::NodeHandle n;
  ros::ServiceServer service =
         n.advertiseService(
         "add_two_ints", add);
  ROS_INFO("Ready to add two ints.");
  ros::spin();
  return 0;
}
```

Client in C++

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
#include <cstdlib>

int main(int argc, char **argv) {
  ros::init(argc, argv, "add_two_ints_client");
  if (argc != 3) {
    ROS_INFO("usage: add_two_ints_client X Y");
    return 1;
  }
  ros::NodeHandle n;
  ros::ServiceClient client = n.serviceClient
        <beginner_tutorials::AddTwoInts>(
        "add_two_ints");
```

Client in C++

```
beginner_tutorials::AddTwoInts srv;
  srv.request.a = atoll(argv[1]);
  srv.request.b = atoll(argv[2]);
  if (client.call(srv)) {
    ROS_INFO("Sum: %ld", (long
                  int)srv.response.sum);
  }
  else {
    ROS_ERROR("Failed to call service
                  add_two_ints");
    return 1;
  }
  return 0;
}
```

CMakeLists.txt

```
add_executable(add_two_ints_server
     src/add_two_ints_server.cpp)
target_link_libraries(add_two_ints_
     server ${catkin_LIBRARIES})
add_dependencies(add_two_ints_server
     beginner_tutorials_gencpp)

add_executable(add_two_ints_client
     src/add_two_ints_client.cpp)
target_link_libraries(add_two_ints_
     client ${catkin_LIBRARIES})
add_dependencies(add_two_ints_client
     beginner_tutorials_gencpp)
```

Test the Service

- Autogenerate code for messages and services

```
$ catkin_make
```

(in the workspace)

- Run the two scripts

```
$ rosrun beginner_tutorials
        add_two_ints_server
$ rosrun beginner_tutorials
        add_two_ints_client 2 6
```

Running Remotely

- One master (running `roscore`)
- All nodes must use the same master

```
export ROS_MASTER_URI=
    http://<master_name>:11311
```

- Bi-directional connectivity of all machines on all ports
  - `ping <host_name>`
  - `netcat [ -l | <host_name> ] <port_no>`
- All machines must advertise themselves by resolvable name

Nodelets

- Nodelets may share memory
- Run within NodeletManager (often embedded in running nodes)

  ```
  rosrun nodelet nodelet manager
  __name:=nodelet_manager
  ```
- Launching via nodelet executable

  ```
  rosrun nodelet nodelet load
    nodelet_tutorial_math/Plus
    nodelet_manager __name:=nodelet1
    nodelet1/in:=foo _value:=1.1
  ```
- Test using `rostopic pub/echo`

Nodelets via Launcher

```
<launch>
  <node pkg="nodelet" type="nodelet"
   name="standalone_nodelet"
   args="manager"/>

  <node pkg="nodelet" type="nodelet"
   name= "Plus" args="load
   nodelet_tutorial_math/Plus
   standalone_nodelet">
    <remap from="/Plus/out" to=
      "remapped_output"/>
  </node>
```

(remapped output)

Nodelets via Launcher

```
  <node pkg="nodelet" type="nodelet" name="Plus2"
       args="load nodelet_tutorial_math/Plus
       standalone_nodelet">
    <rosparam file="$(find nodelet_tutorial_math)
          /plus_default.yaml"/>
```

(read value from file)

```
  </node>
  <node pkg="nodelet" type="nodelet" name="Plus3"
   args="standalone nodelet_tutorial_math/Plus">
    <param name="value" type="double"
          value="2.5"/>
    <remap from="Plus3/in" to="Plus2/out"/>
```

(value given and chaining of plus2/3)

```
  </node>
</launch>
```

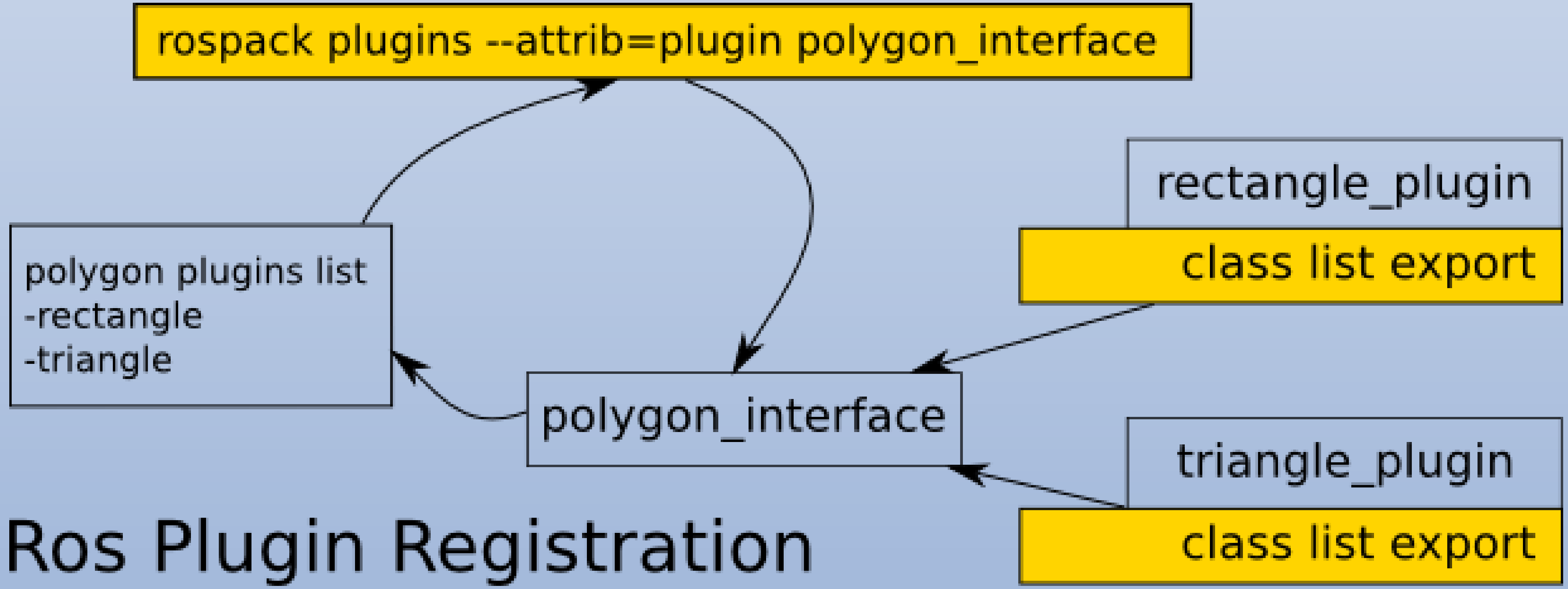
From Node to Nodelet

- add the necessary #includes
- get rid of int main()
- subclass nodelet::Nodelet
- move code from constructor to onInit()
- add the PLUGINLIB_EXPORT_CLASS macro
- add <build_depend> and <run_depend> dependencies on nodelet in the package manifest.
- add the <nodelet> item in the <export> part of the package manifest
- create the .xml file to define the nodelet as a plugin
- make the necessary changes to CMakeLists.txt (comment out a rosbuild_add_executable, add a rosbuild_add_library)

Plugins

- dynamically loadable classes
- loaded from runtime library (shared object, dynamically linked library)
- application not explicitly linked against the library containing the classes
- open library containing exported classes without the application being aware of the library or the header file
- useful for extending/modifying application behavior without needing the application source code

Example


rospack plugins --attrib=plugin polygon_interface
rectangle_plugin
class list export
polygon plugins list
-rectangle
-triangle
polygon_interface
triangle_plugin
class list export
Ros Plugin Registration

polygon_base.h

```
#ifndef PLUGINLIB_TUTORIALS__POLYGON_BASE_H_
#define PLUGINLIB_TUTORIALS__POLYGON_BASE_H_

namespace polygon_base {
  class RegularPolygon {
    public:
      virtual void initialize(double side_length) = 0;
// initialize required for non-standard constructors
      virtual double area() = 0;
      virtual ~RegularPolygon(){}
    protected:
      RegularPolygon(){}
  };
};
#endif
```

polygon_plugins.h

```
#ifndef PLUGINLIB_TUTORIALS__POLYGON_PLUGINS_H_
#define PLUGINLIB_TUTORIALS__POLYGON_PLUGINS_H_
#include <pluginlib_tutorials/polygon_base.h>
#include <cmath>

namespace polygon_plugins {

  class Triangle : public polygon_base::RegularPolygon {

    public:
      Triangle(){}
      void initialize(double side_length) {
        side_length_ = side_length;
      }
      double area() {
        return 0.5 * side_length_ * getHeight();
      }
      double getHeight() {
        return sqrt((side_length_ * side_length_)
        - ((side_length_ / 2) * (side_length_ / 2)));
      }
```

polygon_plugins.h

```
    private:
      double side_length_;
  };

  class Square : public
                polygon_base::RegularPolygon {
    public:
      Square(){}
      void initialize(double side_length) {
        side_length_ = side_length;
      }
      double area() {
        return side_length_ * side_length_;
      }
    private:
      double side_length_;
  };
};

#endif
```

polygon_plugins.cpp

```
#include <pluginlib/class_list_macros.h>
#include
    <pluginlib_tutorials/polygon_base.h>
#include
    <pluginlib_tutorials/polygon_plugins.h>


PLUGINLIB_EXPORT_CLASS(polygon_plugins::
    Triangle, polygon_base::RegularPolygon)
PLUGINLIB_EXPORT_CLASS(polygon_plugins::
    Square, polygon_base::RegularPolygon)
```

Activate Plugin

- Add to `CMakeLists.txt`:

```
add_library(polygon_plugins
            src/polygon_plugins.cpp)
```

- Run `catkin_make`
- Instance of plugins can now be created by loading the library
- Plugin loader needs to know about the library and what to reference within it
- Create an XML file that makes the necessary information about plugins available

polygon_plugins.xml

```
<library path="lib/libpolygon_plugins">
  <class type="polygon_plugins::Triangle"
      base_class_type="polygon_base::
      RegularPolygon">
    <description>This is a triangle
         plugin.</description>
  </class>
  <class type="polygon_plugins::Square"
      base_class_type="polygon_base::
      RegularPolygon">
    <description>This is a square
         plugin.</description>
  </class>
</library>
```

Explanation

- `type`: The fully qualified type of the plugin.
- `base_class`: The fully qualified base class type for the plugin.
- `description`: A description of the plugin and what it does.
- `name`: This refers to the name of the plugin (plugin_namespace/PluginName), optional.

Final Steps

- Add to `package.xml`:

  ```
  <export>
    <pluginlib_tutorials plugin=
    "${prefix}/polygon_plugins.xml" />
  </export>
  ```

- The name of the tag (`pluginlib_tutorials`) corresponds to the package where the *base_class* for the plugin lives. In most real-world cases not the same as for the inherited plugin classes.
- Verify that things are working:

  ```
  rospack plugins --attrib=plugin
                   pluginlib_tutorials
  ```

- You should see output giving the full path to the `polygon_plugins.xml` file.

Use in Node

```
#include <pluginlib/class_loader.h>
#include <pluginlib_tutorials/polygon_base.h>

int main(int argc, char** argv) {
  pluginlib::ClassLoader<polygon_base::RegularPolygon>
        poly_loader("pluginlib_tutorials_",
        "polygon_base::RegularPolygon");
  try {
    boost::shared_ptr<polygon_base::RegularPolygon>
        triangle = poly_loader.createInstance(
        "polygon_plugins::Triangle");
    triangle->initialize(10.0);
    ROS_INFO("Triangle area: %.2f", triangle->area());
  }
  catch(pluginlib::PluginlibException& ex) {
    ...
  }
  return 0;
}
```

MyNodeletClass.h

```
#include <nodelet/nodelet.h>

namespace example_pkg {

    class MyNodeletClass :
        public nodelet::Nodelet {
        public:
            virtual void onInit();
    };
}
```

MyNodeletClass.cpp

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

PLUGINLIB_EXPORT_CLASS(example_pkg::
      MyNodeletClass, nodelet::Nodelet)
// capitalization !!

namespace example_pkg {
    void MyNodeletClass::onInit() {
        NODELET_DEBUG("Initializing
                          nodelet...");
    }
}
```

nodelet_plugins.xml

```
<library path="lib/libMyNodeletClass">
  <class
   name="example_pkg/MyNodeletClass"
   type="example_pkg::MyNodeletClass"
   base_class_type="nodelet::Nodelet">
  <description>
  This is my nodelet.
  </description>
  </class>
</library>
```

package.xml

```
...
<build_depend>nodelet</build_depend>
<run_depend>nodelet</run_depend>
<export>
  <nodelet plugin=
   "${prefix}/nodelet_plugins.xml" />
</export>
...
```