Instruction for Preliminary Assignment 2 for Term Project

ROS Structure and Pioneer Control

1. Introduction

For the term project, we are going to use a simulator in ROS which is called "GAZEBO". This simulator runs on ROS framework and communicates with ROS core. So we explained how to send a control to GAZEBO simulator and receive sensing data from it. We used a mobile robot and depth camera in GAZEBO. The mobile robot is Pioneer which can be controlled by sending linear velocity and angular velocity. The depth camera is Kinect XBOX which provides a depth image. The assignment's goal is implementing a control publisher and a depth image subscriber on ROS framework and manually make a mobile robot follow a given path.

2. Structure of ROS

Same as previous instruction, this section highly referred to a tutorial in the ROS wiki (http://wiki.ros.org/ROS/Tutorials). At first, the instruction explains the file system of ROS. Secondly, it introduces the concept of "Node" which is executable unit. Finally, it shows the way to compile and run your *.cpp file.

2.1. File System

Packages: Packages are the software organization unit of ROS code. Each package can contain libraries, executables, scripts, or other artifacts. Actually, you already meet this concept

Manifest (package.xml): A manifest is a description of a *package*. Its serves to define dependencies between *packages* and to capture meta-information about the *package* like version, maintainer, license, etc...

2.2. Understanding ROS Nodes

2.2.1. Nodes

A node really isn't much more than an executable file within a ROS package. ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to a Topic. Nodes can also provide or use a Service.

2.2.2. Client Libraries

ROS client libraries allow nodes written in different programming languages to communicate:

- rospy = python client library,
- roscpp = c++ client library,

when you make new package you have to add rospy and roscpp as dependency.

2.2.3. roscore

`roscore` is the first thing you should run when using ROS.

Please run:

```
$ roscore
```

2.2.4. Using rosrun

`rosrun` allows you to use the package name to directly run a node within a package (without having to know the package path. Actually, you do this at pervious assignment.).

Usage:

```
$ rosrun [package_name] [node_name]
```

You can also check node currently running, following command, rosnode.

In a **new terminal**:

```
$ rosnode list
```

You will see something similar to:

```
/rosout
```

3. Understanding ROS Topics

3.1. ROS Topics

Nodes are communicating with each other over a ROS **Topic**. There are two types of the topic. The one is a **publisher** which publishes messages on a topic. Another is a **subscriber which subscribes messages**. Figure 1 explains this structure. The two nodes communicating each other should share a same topic.

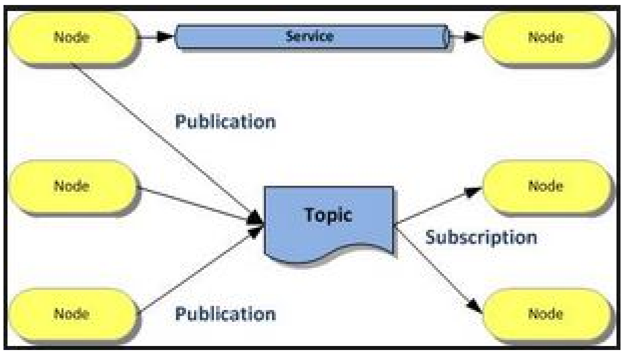


Figure 1. Structure of ROS

There is another method for the communication between nodes, called as 'service'. But we don't use 'service' in this project.

Declaration of publisher

```
ros::NodeHandle n;
ros::Publisher pub = n.advertise<template T>("/topic_name", int que_size);
```

Declaration of subscriber

```
ros::NodeHandle n;
ros::Subscriber sub = n.subscribe("/topic_name", int que_size, Callback);
void Callback(T::ConstPtr msgs){}
```

The node handler 'n' is usually used when we want to make a publisher or a subscriber. The advertise function which makes a publisher instance is a member function of the node handler. The inputs of advertise function are the name of topic and the que size of which default value is usually 1. You also have to set template T as a type of message. 'Subscribe' is also the member function of node handler and it makes subscriber instance. Its input arguments are a name of the topic, a size of the que and a callback function. The callback function is a function.

3.1.1. Using rostopic

Rostopic can show the list of currently running topic and the data published on a topic.

Usage:

```
rostopic list
rostopic echo [topic]
```

3.1.2. Example

(1) Writing the Publisher Node

Create the ~/catkin_ws/src/beginner_tutorials/src/talker.cpp file within the beginner_tutorials (you made it before) package and paste the following inside it:

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

#include <sstream>

/**
 * This tutorial demonstrates simple sending of messages over the ROS system.
 */
int main(int argc, char **argv)
{
  /**
   * The ros::init() function needs to see argc and argv so that it can perform
   * any ROS arguments and name remapping that were provided at the command line. For
programmatic
   * remappings you can use a different version of init() which takes remappings
   * directly, but for most command-line programs, passing argc and argv is the easiest
   * way to do it. The third argument to init() is the name of the node.
   *
   * You must call one of the versions of ros::init() before using any other
   * part of the ROS system.
   */
  ros::init(argc, argv, "talker");

  /**
   * NodeHandle is the main access point to communications with the ROS system.
   * The first NodeHandle constructed will fully initialize this node, and the last
   * NodeHandle destructed will close down the node.
   */
  ros::NodeHandle n;

  /**
   * The advertise() function is how you tell ROS that you want to
   * publish on a given topic name. This invokes a call to the ROS
   * master node, which keeps a registry of who is publishing and who
   * is subscribing. After this advertise() call is made, the master
   * node will notify anyone who is trying to subscribe to this topic name,
   * and they will in turn negotiate a peer-to-peer connection with this
   * node.  advertise() returns a Publisher object which allows you to
   * publish messages on that topic through a call to publish().  Once
   * all copies of the returned Publisher object are destroyed, the topic
   * will be automatically unadvertised.
   *
   * The second parameter to advertise() is the size of the message queue
   * used for publishing messages.  If messages are published more quickly
   * than we can send them, the number here specifies how many messages to
   * buffer up before throwing some away.
   */
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);

  ros::Rate loop_rate(10);

  /**
   * A count of how many messages we have sent. This is used to create
   * a unique string for each message.
   */
  int count = 0;
  while (ros::ok())
  {
    /**
     * This is a message object. You stuff it with data, and then publish it.
     */
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());

    /**
     * The publish() function is how you send messages. The parameter
     * is the message object. The type of this object must agree with the type
     * given as a template parameter to the advertise<>() call, as was done
     * in the constructor above.
     */
    chatter_pub.publish(msg);

    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }


  return 0;
}
```

(2) Writing the Subscriber Node

Create the ~/catkin_ws/src/beginner_tutorials/src/listener.cpp file within the beginner_tutorial pachage and paste the following inside it.

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

/**
 * This tutorial demonstrates simple receipt of messages over the ROS system.
 */
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  /**
   * The ros::init() function needs to see argc and argv so that it can perform
   * any ROS arguments and name remapping that were provided at the command line. For
programmatic
   * remappings you can use a different version of init() which takes remappings
   * directly, but for most command-line programs, passing argc and argv is the easiest
   * way to do it. The third argument to init() is the name of the node.
   *
   * You must call one of the versions of ros::init() before using any other
   * part of the ROS system.
   */
  ros::init(argc, argv, "listener");

  /**
   * NodeHandle is the main access point to communications with the ROS system.
   * The first NodeHandle constructed will fully initialize this node, and the last
   * NodeHandle destructed will close down the node.
   */
  ros::NodeHandle n;

  /**
   * The subscribe() call is how you tell ROS that you want to receive messages
   * on a given topic. This invokes a call to the ROS
   * master node, which keeps a registry of who is publishing and who
   * is subscribing. Messages are passed to a callback function, here
   * called chatterCallback. subscribe() returns a Subscriber object that you
   * must hold on to until you want to unsubscribe. When all copies of the Subscriber
   * object go out of scope, this callback will automatically be unsubscribed from
   * this topic.
   *
   * The second parameter to the subscribe() function is the size of the message
   * queue. If messages are arriving faster than they are being processed, this
   * is the number of messages that will be buffered up before beginning to throw
   * away the oldest ones.
   */
  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

  /**
   * ros::spin() will enter a loop, pumping callbacks. With this version, all
   * callbacks will be called from within this thread (the main one). ros::spin()
  * will exit when Ctrl-C is pressed, or the node is shutdown by the master.
  */
 ros::spin();

 return 0;
}
```

(3) Building and Running your nodes

You used catkin_create_pkg in the previous tutorial which created a package.xml and a CMakeLists.txt file.

The generated CMakeLists.txt should look like this (with modifications from the Creating Msgs and Srvs tutorial and unused comments and examples removed):

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


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

You did this at previous assignment. Compile the source files.

```
# In your catkin workspace
$ catkin_make
```

After compilation, make sure that a roscore is up and running. Run nodes.

```
$ roscore
$ rosrun beginner_tutorials listener (in new terminal)
$ rosrun beginner_tutorials talker (in new terminal)
```

4. GAZEBO simulator

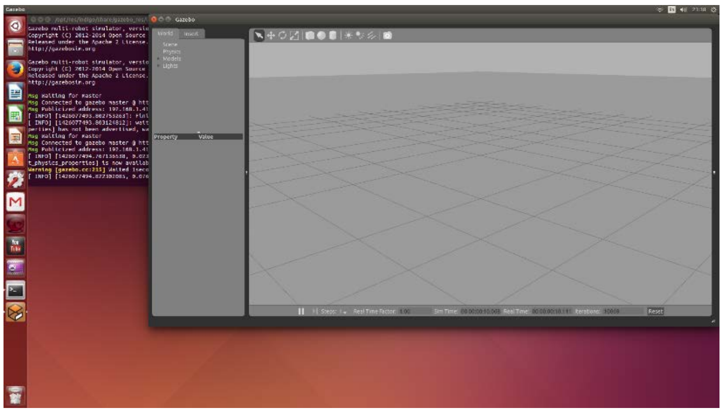
You already have a GAZEBO simulator. When you installed ROS indigo, GAZEBO was installed together. So you can easily start the program. This section explains the way to use a gazebo and communicate with a gazebo world.

4.1. Using roslaunch to Open World Models

The roslaunch tool is the standard method for starting ROS nodes and bringing up robots in ROS. To start an empty Gazebo world similar to the rosrun command in the previous tutorial, simply run this.

```
roslaunch gazebo_ros empty_world.launch
```

You can see the gazebo world like figure 2.



```
Gazebo multi-robot simulator, versio
Copyright (C) 2012-2014 Open Source
Released under the Apache 2 License.
http://gazebosim.org

Gazebo multi-robot simulator, versio
Copyright (C) 2012-2014 Open Source
Released under the Apache 2 License.
http://gazebosim.org

Msg Waiting for master
Msg Connected to gazebo master @ htt
Msg Publicized address: 192.168.1.41
[ INFO] [1426077493.802753263]: Fini
[ INFO] [1426077493.803124812]: Wait
perties] has not been advertised, wa
Msg Waiting for master
Msg Connected to gazebo master @ htt
Msg Publicized address: 192.168.1.41
[ INFO] [1426077494.767136538, 0.023
t_physics_properties] is now availab
Warning [gazebo.cc:215] Waited 1seco
[ INFO] [1426077494.822382085, 0.076
```

Figure 2. GAZEBO Empty world

In your project file, there is a launch file to open a world model which contains the robot 'pioneer' and the Kinect.

5. Assignment

This assignment is making nodes to control pioneer and receive Kinect depth image. This will be helpful for your future project assignment. You have to download 'pre_assignment2.tar.gz' file and extract it at ~/catkin_ws/src/. Before compilation, you must install ros-indigo-joy package. It can be downloaded by package manager, using "sudo apt-get install ros-indigo-joy". After installation, compile your project. You will make a controller and an image viewer for a depth image.

5.1. GAZEBO world

You can launch the GAZEBO world using a following command.

```
roslaunch pre_assignment2 pre_assignment2.launch
```

You will see walls, tables and a pioneer with the Kinect sensor. Check which node is running by using rosnode command. After that, check which topic is made by the node using rostopic command.

```
rosnode list
rostopic list
```

There are a lot of topics and nodes. But we only use two topics:

"/camera/depth/image_raw" and "/RosAria/cmd_vel".

5.2. Camera Topic

In the GAZEBO, there exists a node that publishes a depth image to "/camera/depth/image_raw" topic. So this node is a publisher. If you made a subscriber of the topic, you can receive the depth_image from the topic. This can be manually checked by the following command.

```
rosrun image_view image_view image:="/camera/depth/image_raw"
```

You can see the window like figure 3.

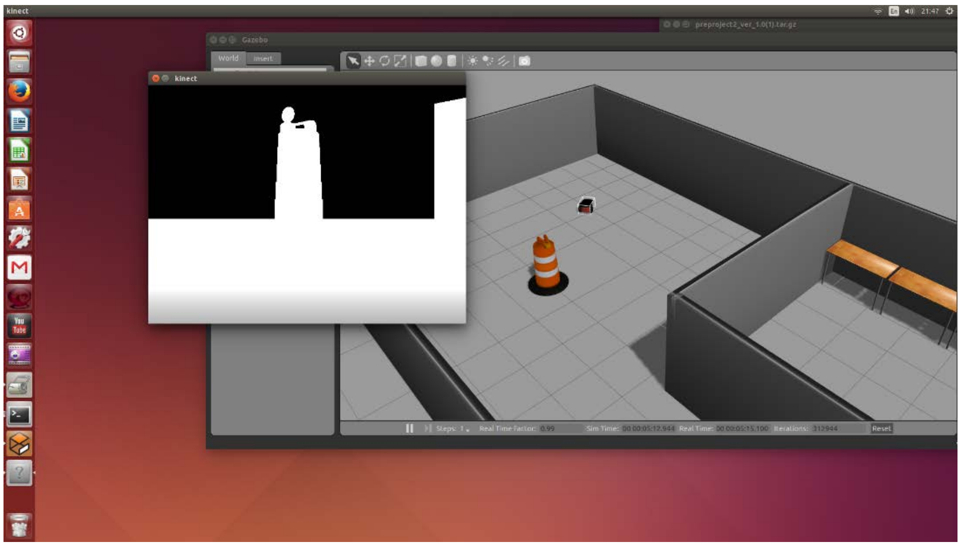


Figure 3. Depth Image Viewer

5.3. Velocity Topic

In the GAZEBO, there exists a node to subscribes a velocity command from "/RosAria/cmd_vel" topic. So this node is a subscriber. If you made a publisher of the topic, you can send the velocity of the pioneer. This node will be a controller of the pioneer.

6. Skeleton Code

- ✓ kinect_viewer.cpp
- ✓ pioneer_manual_controller.cpp

6.1. kinect_viewer.cpp

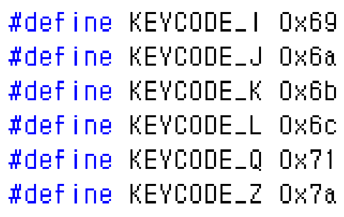
Fill in the TODO part. Write a code of subscriber that gets message from "/camera/depth/image_raw" topic and then call Kinect_callback function. You don't need to modify any other part of the skeleton code.

6.2. pioneer_manual_controller.cpp

This node publishes a pioneer control to the /RosAria/cmd_vel topic. You need to write a code which publishes an adequate control of the pioneer. The control of the pioneer has two velocities: linear and angular. Control varies with the keyboard input so that you can control the pioneer as you want in the GAZEBO when executing the node. Following table is pair of a keyboard input and a control output. The units of the linear velocity and the angular velocity are m/s and rad/sec, respectively.

i	Go straight without rotation	**,**	Go backward without rotation
j	Turn left at a fixed position	**O**	Move forward with right turn
l	Turn right at a fixed position	**U**	Move forward with left turn
m	Move backward with clockwise rotation	**.**	Move backward with counter clockwise rotation
k	Stop		
q	Increase angular and linear speeds by 10%	**Z**	Decrease angular and linear speeds by 10%
w	Increase linear speed by 10%	**X**	Decrease linear speed by 10%
e	Increase angular speed by 10%	**C**	Decrease angular speed by 10%

We defined ASCII code of keyboard inputs at the top of skeleton code like figure 4. You can see the variable "keyboard_input" in your skeleton code. This variable has a keyboard input value. You can check which key is pressed watching "keyboard_input" with ASCII code.



```
#define KEYCODE_I 0x69
#define KEYCODE_J 0x6a
#define KEYCODE_K 0x6b
#define KEYCODE_L 0x6c
#define KEYCODE_Q 0x71
#define KEYCODE_Z 0x7a
```

Figure 4. ASCII code

In the skeleton code, there exists a member variable "cmdvel" whose type is geometry_msgs::Twist. You have to send this argument to "/RosAria/cmd_vel". The publisher is also declared as member variable, "pub_". You use this variable to publish "cmdvel". 'geometry_msgs::Twist' type has two member variables: "linear" and "angular". "linear" means the linear velocity and "angular" means the angular velocity. "linear" is a 3 dimensional vector and you can just use x-axis cause the pioneer can't directly move towards the y-axis and the z-axis. "angular" is a 4 dimensional vector which is called "quaternion". In our case, since our pioneer rotates in the ground plane, you can just use the z-value. The others must be set to 0. For example:

```
cmdvel.linear.x = [linear velocity];

cmdvel.linear.y = 0;

cmdvel.linear.z =0;

cmdvel.angular.x = 0;

cmdvel.angular.y = 0;

cmdvel.angular.z = [angular velocity];

cmdvel.angular.w = 0;
```

Controller should be set the cmdvel like this and send it to "RosAria/cmd_vel" topic.

7. Submission Format

Compress your project folder which includes all your project files. Then, upload it on eTL. The name of the compressed file should be "**IS2016_[StudentNumber]_preassign02.zip**". For example, if your ID number is 123456789, the file name should be 'IS2016_123456789_preassigin02.zip'.