ROBT502
Robot Perception & Vision

Lab Series

Instructor:

Zhanat KAPPASSOV, Assistant Professor

Assistants:

Saltanat SEITZHAN, MS student

Nurlan KABDYSHEV, MS student

Amina ASREPOVA, Coordinator

Attendance is compulsory

- Fill the attendance sheet
- Report will not be accepted if you are absent

Bootable memory stick

- How to use BOOTABLE flash drive is on Moodle
- Bootable_flash_Linux_ROS.pdf

Git for uploading your code

- Git tutorial (on Moodle)
- Git tutorial on web (https://www.atlassian.com/git/tutorials):
  - https://www.atlassian.com/git/tutorials/install-git
  - https://www.atlassian.com/git/tutorials/setting-up-a-repository
  - https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init

What is ROS?

- ROS is an open-source, meta-operating system used for robotics systems. It stands for Robotic Operating System
- As the name suggests, it provides all the features that one would expect from an OS:
  - Package management
  - Low-level device control
  - Hardware abstraction
  - Implementation of commonly-used functionality
  - **Message-passing between processes**

Applications of ROS

- Simulation and visualization: rviz, stage (2D), Gazebo engine.
- Drivers: camera_drivers, laser_drivers, imu_drivers.
- 3D processing: point_cloud_perception (PCL), laser_pipeline.
- Image Processing: image_pipeline, vision_opencv (OpenCV).
- 3D Transformations: tf, tf_conversions.
- Navigation: SLAM, collision detections.
- Controllers: pr2_controller_manager.
- Robot modelling and Kinematics-Dynamics: urdf, KDL.
- Motion Planning: algorithms (OMPL, SBPL, CHOMP, etc.).
- Grasping/Manipulation: OpenRAVE, MoveIt.

ROS File management system

- **Packages**: main component of ROS's software organization. A package may include datasets, configuration files, ROS runtime processes (referred to as "nodes"), a ROS-dependent library, and other items that are logically grouped together.
- **Package Manifests**: a file named package.xml that provides all metadata about a package – name, version, description, license, dependencies

Package typical structure

- My_Package/
  - include/package_name - header files C++
  - msg/ - different message types, define the data structures for messages sent within ROS
  - srv/ - services, define the request and response data structures for services in ROS.
  - src/ - C++ files
  - scripts/ - Python files
  - **CMakeLists.txt** – a file to compile the package in a CMake manner
  - **package.xml** – manifest of a package

ROS Computation Graph

- What is Computation Graph?
  - It is the network of processes in ROS that are processing data. These processes can “talk” to each other via messages or requests.
- **Node**
  - It is a process that performs some computations. The paradigm of ROS is to for system to be modular, i.e a system usually comprises many nodes
  - For example, for a control of a mobile platform, one node would control the wheels’ motors, another would be processing laser data, the third would do the motion planning, the fourth localization and etc.
  - A ROS node is written with the help of a ROS client library: roscpp and rospy

ROS Computation Graph

- **Master**
  - The master node, the one which provides name registration, and network for different nodes to communicate or invoke services.
  - If master fails, then the whole network fails, the big limitation that was fixed in ROS2
- **Messages**
  - Data structure that nodes are using to communicate
  - Comprise primitive types – integer, float, Boolean, string, arrays, etc.
  - Messages can include arbitrarily nested structures and arrays (much like C structures)

ROS Computation Graph

- **Topics**
  - A building unit of Publish/Subscribe transport system that is used in ROS (look up in the internet other systems)
  - The topic is a name that is used to identify the content of the message
  - Basically, a node sends out a message by publishing it to a specific topic
  - A node that is interested in a certain kind of data will subscribe to the appropriate topic.
  - Initial connection between subscribers and publishers by Master. Then, peer-to-peer TCP (Transmission Control Protocol) communication.

ROS Computation Graph

- **Services**
  - Although the Publish/Subscribe paradigm is immensely versatile, Request/Reply interactions—which are frequently needed in distributed systems—do not work well with its many-to-many, one-way transport.
  - Request/Reply is done via services
  - It’s defined by a pair of message structures: one for the request and one for the reply
  - A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply

ROS Computation Graph

- **Services**

There are two parts in a srv file, separated by “---”:

Request

Response

#request

int8 foobar

another_pkg/AnotherMessage nomVariable1

#response

another_pkg/YetAnotherMessage nomVariable2

uint32 an_integer

Automatic generation: “generate_messages()” in CMakeLists.txt

ROS – Basics of Node

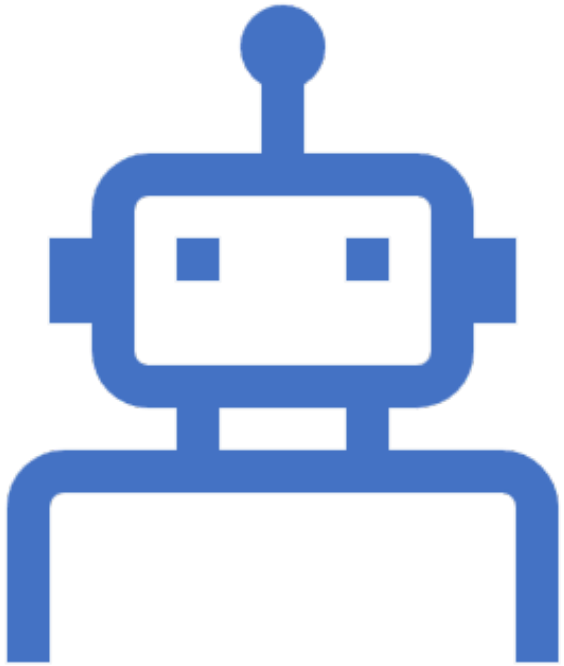
- **roscore** – running a master, first thing that is needed to be run
- **rosnode** - displays information about the ROS nodes that are currently running
  - **rosnode list** – lists all the active nodes
  - **rosnode info /[node_name]** – gives information about a specific node
- **rosrun** - allows you to use the package name to directly run a node within a package
  - **rosrun [package_name] [node_name]**
  - **rosrun turtlesim turtlesim_node**

ROS Basics

- Let’s start with creating a workspace for our project
  - **mkdir –p catkin_ws/src**
  - **cd catkin_ws**
  - **catkin_make** - Catkin is the official build system of ROS and the successor to the original ROS build system, rosbuild
- You always need to source setup.bash file that is located in catkin_ws/devel/setup.bash
  - Either write **source devel/setup.bash** every time you open a new terminal
  - Or add “**source [ws_path]/devel/setup.bash**” to **.bashrc** file
    - gedit ~/.bashrc
    - And add it manually

ROS Basics

Create	Create your own package • catkin_create_pkg [name] [dependencies] • catkin_create_pkg my_package std_msgs roscpp rospy
Build	Build the workspace • catkin_make (in the catkin_ws/src directory)
Refresh	Refresh the package list • rospack profile
Don’t forget	Don’t forget to source your .bash file • source devel/setup.bash



Todays lab - Lab 1

- Navigate within Linux (Ubuntu – it is a Linux distribution based on **Debian OS**)
- Robot Operating System (ROS) (http://wiki.ros.org/kinetic/Installation/Ubuntu) and Use any Editor or **IDE** for writing codes (it is *gedit* by default, *Kdevelop* could be installed)

*Understand Cmake, make, gcc (https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.ht ml and https://cmake.org/cmake/help/latest/guide/tutorial/index.html)

Digression: open source does not mean nobody gets paid: there is **redhat** company https://www.redhat.com/en/enterprise-linux-8

Lab 1 tasks and files to submit

- Perform first 16 steps in ROS Tutorials (http://wiki.ros.org/ROS/Tutorials)
  - Exercise 0: Create the publisher that sends every time an Integer value and that has the topic name with your surname (step 11 or 12 (any) in the tutorial). Publish the numbers of your NU id card in the loop at every execution ros:spin() once per second and then 100 times per second). For this you need to create a Publisher and Subscriber with the Topic name of your name that sends the digits of your NU id numbers one by one in the loop at 1 Hz rate and 50Hz rate.

- Perform the next five exercises.

Report
- Introduction on ROS and your own examples of robots that use ROS.
- Screenshots of exercises zero to five.
- Code.

Exercise 1

- Now, we have workspace, and we have our package. Let’s create two nodes that would communicate with each other
- Move to *catkin_ws/src/my_package/src*
- Create a file and name it **publisher.cpp**
- Copy and paste the code from file given to you to your file
- Do the same with a new file and name it **subscriber.cpp**

Exercise 1

- Okay, how to run these nodes? First we need to modify **CMakeLists.txt** file
  - Open **CMakeLists.txt**
  - Check that **find_package** looks for all dependencies
  - Add your cpp node as an executable and link with the required libraries

    add_executable(talker src/publisher.cpp)
    target_link_libraries(talker ${catkin_LIBRARIES})
    add_executable(listener src/subscriber.cpp)
    target_link_libraries(listener ${catkin_LIBRARIES})
- Finally, compile the workspace with **catkin_make**

Exercise 1

- Now, we have two nodes compiled and ready to run
- From the first terminal
  - **rosrun my_package talker**
- From the second terminal
  - **rosrun my_package listener**
- Now, if everything is correct, we can see two nodes talk to each other via ROS Network

Exercise 2

- Now, lets work with a good example of a robot **turtlesim**
  - Create a new package
  - **catkin_create_pkg turtlebot_controller turtlesim std_msgs geometry_msgs roscpp rospy**
- Create a **subscriber.cpp** file in the **src/** folder
- Now, we will write code that would subscribe to the **turtlesim** node and show us its position

Exercise 2

- Study the file given to you **turtle_listener.cpp**
  - How can I determine the message's structure and type based on the topic?
- For that we need to start the node first
  - **rosrun turtlesim turtlesim_node**
- Now, we can use ROS tools to explore the topics this node is publishing
  - rostopic info /turtle1/pose
  - rosmsg show turtlesim/Pose
- If there is an error with turtle_listener.cpp when building the workspace (catkin_make), try to delete and retype the quotation marks in #include lines around packages

Exercise 2

What is a callback function?

It is a function that is called every time a new message is available

If you called ros::spin() as we did at the end of the code, the callback function will be running in the thread and blocking the process of finishing

So, in our case, every time a new message is published on the turtle1/pose topic the callback function is called

Note

The message has been passed in a boost_shared_ptr and member of the class being pointed to can be accessed using the dereferencing operator ‘->’

Exercise 2

- Again add the executable and link libraries to the CMakeLists.txt
- Compile
- And run
  - **roscore**
  - **rosrun turtlesim turtlesim_node**
  - **rosrun turtlebot_controller turtle_listener**
- You should see the position and orientation of the turtle being printed into the terminal

Exercise 3

- Okay, lets move on and control the turtle from our code
- First, identify what topic and what kind of message is needed
  - **rostopic list –v** <---- this displays a verbose list of available topics
  - We can see a topic **/turtle1/cmd_vel ---** cmd_vel stands for command velocities, so in order to control the turtle we have to send set of velocities to this topic.
  - We can also see that a message required to publish to this topic is of type of **geometry_msgs/Twist**
  - **rosmsg show geometry_msgs/Twist ---** look at the entries of the message
- Let’s add it to our **turtle_listener.cpp**

Exercise 3

- So, the message contains these entries, as we can see we have to specify linear and angular velocities. As the turtle is planar, then in order to move along the plane, we only need to specify **linear.x** and **linear.y** velocities. And in order to turn left or right **angular.z** velocity

```
•geometry_msgs/Vector3 linear
    •float64 x
    •float64 y
    •float64 z
•geometry_msgs/Vector3 angular
    •float64 x
    •float64 y
    •float64 z
```

Exercise 3

- Okay, how do we define it in code?
  - First add header file of the message we need to use
    - ***#include “geometry_msgs/Twist.h”***
  - As we need to publish, then we have to define a publisher
    - ***ros::Publisher pub;***
  - Now, specify to which topic to publish and with what message
    - ***pub = nh.advertise<geometry_msgs::Twist>(“turtle1/cmd_vel”, 1);***
  - So, from previous slides we know that callback function is called every time new message to the pose topic is arrived, so we can add our code to the callback function, and it will be executed as well

Exercise 3

- Add this snap of code into the callback function
  - ***geometry_msgs::Twist my_vel;***
  - ***my_vel.linear.x = 1.0;***
  - ***my_vel.angular.z = 1.0;***
  - ***pub.publish(my_vel);***
- Compile the workspace and run all the nodes again. Now you should see the turtle moving

Exercise 4

- As we could see Publish/Subscribe architecture is very useful. However, it does not suit for RPC (Remote Procedure Call) - Request/Reply Interaction, which are heavily used in many distributed systems
- In ROS it can be done via **Services**
- Let’s see how we can use them
  - **rosservice** or **rossrv**
    - **rosservice list** --- lists all available services
    - **rosservice call [name] [args]** --- calls the service with given arguments
    - **rosservice type [name]** --- prints info about the service

Exercise 4

- For example, let’s see what available services we have with turtlesim
  - **rosservice list**
    - **/clear**
    - **/kill**
    - **/reset**
    - **/rosout/get_loggers**
    - **/rosout/set_logger_level**
    - **/spawn /teleop_turtle/get_loggers**
    - **/teleop_turtle/set_logger_level**
    - **/turtle1/set_pen**
    - **/turtle1/teleport_absolute**
    - **/turtle1/teleport_relative**
    - **/turtlesim/get_loggers**
    - **/turtlesim/set_logger_level**
  - ***/clear*** clears the background of the turtlebot environment, let’s call it, but first we need to see what arguments it takes

Exercise 4

- ***rosservice type /clear***
  - **std_srvs/Empty ---** it means it takes no arguments, i.e. it sends no data when making a **request** and receives no data when receiving a **response**
- Now, we can call it
  - ***rosservice call /clear***
  - And it should clear the background
- Okay, let’s try to use **/spawn** service, this service spawns a new turtle in the environment. Again, first check what arguments it takes

Exercise 4

- Our aim is to add to the code interaction with services
- Again, open the .cpp file
  - First, add ***“turtlesim/Spawn.h”*** header
  - Create a client that sends requests to the specific service
    - ***ros::ServiceClient client1 = nh.serviceClient<turtlesim::Spawn>(“/spawn”);***
- You should have already studied what is the structure of the service of **/spawn**
  - ***turtlesim::Spawn srv1;*** --- define a /spawn service message
    - ***srv1.request.x = 1.0;***
    - ***srv1.request.y = 5.0;***
    - ***srv1.request.theta = 0.0;***
    - ***srv1.request.name = “Turtle_YOURNAME”***
- And, finally, call the service
  - ***client1.call(srv1);***

Final exercise: Exercise 5

- Okay, the exercise is to create a turtlebot and control it to move along a specific trajectory
  - First, kill the **turtle1**
  - Spawn your own turtle named ***turtle_YOURNAME*** at the center of the environment
  - Use service (teleport_absolute service) in order to put your turtle on one of the corners
  - Use topics to control the turtle, such that it moves in a square (from each corner to the next one) and in a triangular trajectory (from first corner move to the second, then to the third, and in diagonal to the first one again)

Thank you for your attention!

- Okay, now we understood how we can use Nodes, Topics, Services. It is a building blocks of all ROS projects. In further labs we will explore RQT, RViz, Gazebo and other ROS tools that would be helpful for your future projects