# Introductory Robot Programming

## ENPM809B

### Lecture 12 – The Robot Operating System (ROS) – Part II

**Zeid Kootbally**

**zeidk@umd.edu**

Fall 2020

# Overview I

## 01 Topics

rostopic

Summary

## 02 Messages

## 03 Summary

## 04 IDE with Catkin Workspace

## 05 Package

Publisher

Subscriber

Publisher/Subscriber

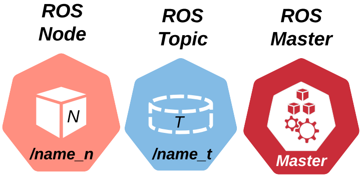
# Conventions

- Slides created with Beamer LaTeX.
- C++code displayed as follows with the LaTeX minted package:

```cpp
#include <iostream>

int main(){
    std::cout << "Hello" << std::endl;
}
```

- Files: file.cpp.
- Directories: *directory*.
- Packages: package.
- `Command to use`
- `Actual command to enter in terminal`



# Highlights

- Prerequisites:
  - Catkin workspace from previous lecture.
  - Ubuntu 18.04 + ROS Melodic.
- This lecture is the second lecture on ROS core components.
- We will write a Publisher and a Subscriber to control robot actuators and read the robot pose, respectively.
- We will learn more about Messages today.
- No ROS assignment.
- Keywords: Topic, Message, Subscriber, Publisher.

# Topics

- A Node is a running instance of a ROS program.
- Nodes can publish Messages to Topics.
- Nodes can subscribe to Topics to retrieve Messages.
- What kind of information is contained in those Messages?
  - Lets take a closer look at Topics and Messages.
  - You can list active Topics with `rostopic list`

```
roslaunch turtlebot3_fake turtlebot3_fake.launch
```

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

```
rostopic list
```

```
/clicked_point
/cmd_vel
/initialpose
/joint_states
/move_base_simple/goal
/odom
/rosout
/rosout_agg
/tf
/tf_static
```

# Topics | rostopic

- To see the type of the Messages published on a Topic:

```
rostopic info <topic_name>
```

```
rostopic info /cmd_vel
```

```
Type: geometry_msgs/Twist

Publishers:
 * /turtlebot3_teleop_keyboard (http://linux:41079/)

Subscribers:
 * /turtlebot3_fake_node (http://linux:39057/)
```

- The name of the package is geometry_msgs and the Message type is Twist, which expresses velocity in free space broken into its linear and angular parts.

# Topics | rostopic

- To see the actual message being published on a topic:

```
rostopic echo <topic_name>
```

```
rostopic echo /cmd_vel
```

- Press on keys to see some of these numbers change.
- Each message is separated with a dashed line.
- The example on the right displays only two messages being published on the topic /cmd_vel

```
---
linear:
  x: 0.2
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: -2.8
---
linear:
  x: 0.2
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: -2.84
---
```

## Topics | Summary

- When we press on some specific keys on the keyboard, messages of type geometry_msgs/Twist are sent to the topic cmd_vel
- Although we can see what the messages look like when they are publish on a topic, we do not know much about the numbers being passed.
- To know more about a message, we can:
  - Check the geometry_msgs package documentation.
  - Use the `rosmsg` tool.

# Messages

- To obtain details on a message type:

```
rosmsg show <message_type>
```

```
rosmsg show geometry_msgs/Twist
```

- Both `linear` and `angular` are composite fields whose data type is `geometry_msgs/Vector3`.
- The indentation shows that `x`, `y`, and `z` are members within those two top-level fields.
- A message with type `geometry_msgs/Twist` contains exactly six numbers, organized into two vectors called `linear` and `angular`.
- Each of these numbers has the built-in type `float64`.

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

- If you are interested, you can look at the Message file directly on your computer:

```
roscd geometry_msgs/msg/
```

```
more Twist.msg
```

```
more Vector3.msg
```

# Messages

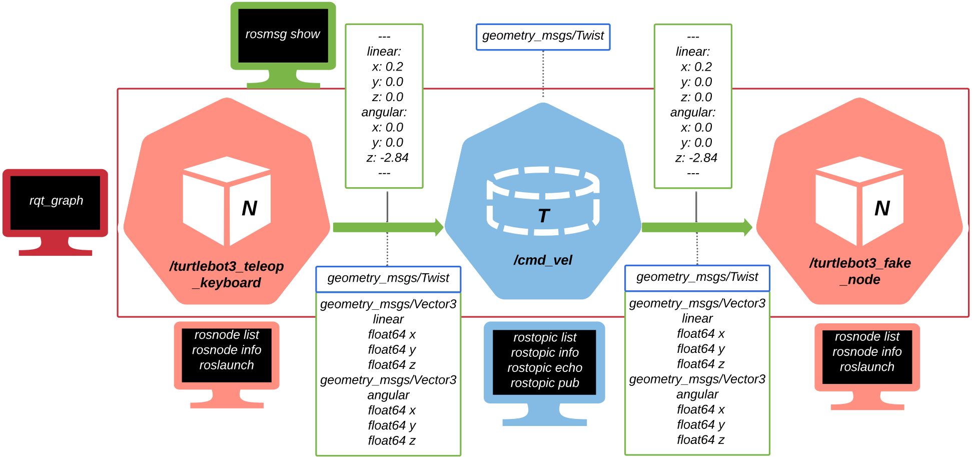
- Most of the time, the work of publishing Messages is done by specialized programs (e.g., C++ or Python code).
- You may find it useful at times to publish Messages by hand.
- This is done with:

```
rostopic pub -r <rate-in-hz> <topic-name> <message-type> <message-content>
```

```
rostopic pub -r 1 cmd_vel geometry_msgs/Twist '[4,0,0]' '[0,0,0]'
```

- This message commands the turtlebot to drive straight ahead (along its x-axis).
- The values are assigned to message fields in the same order that they are shown by `rosmsg show geometry_msgs/Twist`

# Summary



rqt_graph

rosmsg show

/turtlebot3_teleop_keyboard

rosnode list
rosnode info
roslaunch

```
---
linear:
  x: 0.2
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: -2.84
---
```

geometry_msgs/Twist

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

geometry_msgs/Twist

T

/cmd_vel

rostopic list
rostopic info
rostopic echo
rostopic pub

```
---
linear:
  x: 0.2
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: -2.84
---
```

geometry_msgs/Twist

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

/turtlebot3_fake_node

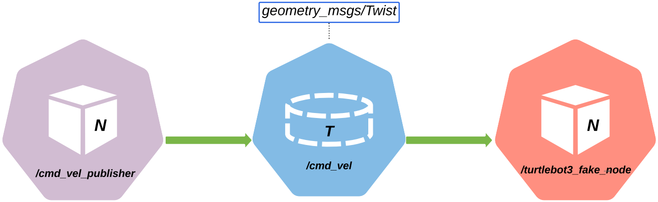
rosnode list
rosnode info
roslaunch

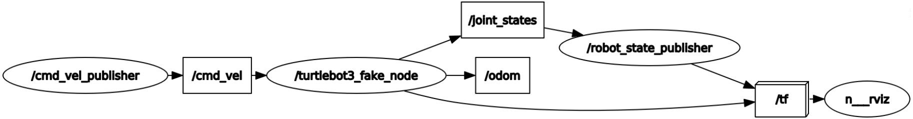
# IDE with Catkin Workspace

- Go to this link to set up CLion to work with your catkin workspace.
- See a list of IDEs compatible with catkin workspace here.

# Package | Publisher

- We want to write a Publisher which will send Messages of type geometry_msgs/Twist to the Topic /cmd_vel.





# Package | Publisher

- We need to create our publisher node in a custom ROS package.
- We need to tell ROS that our package requires the package geometry_msgs, i.e., it depends on geometry_msgs.
- To be able to write C++ code in ROS you also need the roscpp package, which is a pure C++ client library for ROS. The roscpp client API enables C++ programmers to quickly interface with ROS Topics, Services, and Parameters.
- These two packages should come with your ROS installation. Try:

```
rospack find geometry_msgs
```

```
rospack find roscpp
```

# Package | Publisher

- Steps to create a Publisher:
  1. Create a new package velocity_publisher in your workspace. This package has two dependencies: roscpp and geometry_msgs
  2. In *velocity_publisher/src*, create a new file vel_pub.cpp (this file will contain code for our publisher node).
     - You can download this file from Canvas.
  3. Edit CMakeLists.txt and package.xml.
  4. Compile your package with `catkin build` and run your Node with `rosrun`

## Package | Publisher | Create a Package

1. Create a new package velocity_publisher in your workspace. This package has two dependencies: roscpp and geometry_msgs
   - Go to *~/catkin_ws/src*: `cd ~/catkin_ws/src`
   - Create a package:

     ```
     catkin create pkg velocity_publisher --catkin-deps roscpp geometry_msgs
     ```

     ```
     catkin build
     ```
   - Try `roscd velocity` + Tab key: If your package does not show up then try `source ~/catkin_ws/devel/setup.bash`
   - Try again `roscd velocity` + Tab key (suggestions include the new package).

- Reminder: : `source` setup.bash from *catkin_ws/devel* after creating a package.

# Package | Publisher | Write Publisher

2. In *~/catkin_ws/src/velocity_publisher/src*, create a new file vel_pub.cpp
   - Steps for publishing:
     (i) Initialize the ROS system.
     (ii) Advertise that we are going to be publishing geometry_msgs/Twist messages on the /cmd_vel topic to the Master.
     (iii) Loop while publishing messages to /cmd_vel 2 times per second.

- Full C++ tutorial on publisher/subscriber can be found here.

# Package | Publisher | package.xml and CMakeLists.txt

3. Edit CMakeLists.txt and package.xml.
   - CMakeLists.txt is the input to the CMake build system for building software packages. Any CMake-compliant package contains one or more CMakeLists.txt file that describe how to build the code and where to install it to.
   - Description of the different fields in CMakeLists.txt can be found here

   (i) Open CMakeLists.txt

   (ii) Declare a C++ executable:
   add_executable($PROJECT_NAME_node src/vel_pub.cpp)

   (iii) Specify the libraries to use when linking a given target:
   target_link_libraries($PROJECT_NAME_node $catkin_LIBRARIES)

# Package | Publisher | package.xml and CMakeLists.txt

- The package manifest (package.xml) must be included with any catkin-compliant package's root folder.
- This file defines properties about the package such as the package name, version numbers, authors, maintainers, and dependencies on other catkin packages.
- Catkin makes use of it to build the dependency tree to determine build order.
- The `rosdep` tool uses this manifest to install system dependencies required by ROS packages.

  ```
  rosdep install <package_name>
  ```

- You may want to edit the contents for the following tags:
  - `<version>`
  - `<description>`
  - `<maintainer>`
  - `<license>`
  - `<author>`

## Package | Publisher | Compile and Run

4. Compile your package with `catkin build` or `catkin build vel_publisher`

```
catkin build velocity_publisher
```

- Start the turtlebot

```
roslaunch turtlebot3_fake turtlebot3_fake.launch
```

- Run your Node with `rosrun <package> <node>`
  - `rosrun` allows you to run an executable in an arbitrary package from anywhere without having to give its full path or `cd/roscd` there first.

```
rosrun velocity_publisher velocity_publisher_node
```

- Check the status of your node with `rosnode` and `rqt_graph`

## Package | Publisher | Exercise#1

- Get the robot’s name from the parameter server and edit the following code snippet to include the robot’s name.

```
ROS_INFO_STREAM("Sending random velocity command:"
    << " linear=" << msg.linear.x
    << " angular=" << msg.angular.z);
```

- Change it to:

```
ROS_INFO_STREAM("Sending random velocity command to " << robot_name
    << ": linear=" << msg.linear.x
    << " angular=" << msg.angular.z);
```

- See here to get the value of a parameter from the parameter server.

# Package | Publisher | Exercise#2

- Use a launch file to start your custom node cmd_vel_publisher using `roslaunch`
- Create a launch file:

```
cd ~/catkin_ws/src/vel_publisher
```

```
mkdir launch
```

```
touch launch/vel_publisher.launch
```

- Edit the launch file and add a node tag:

```
<launch>
  <node pkg="vel_publisher"
        type="vel_publisher_node"
        name="cmd_vel_publisher"
        output="screen">
  </node>
</launch>
```

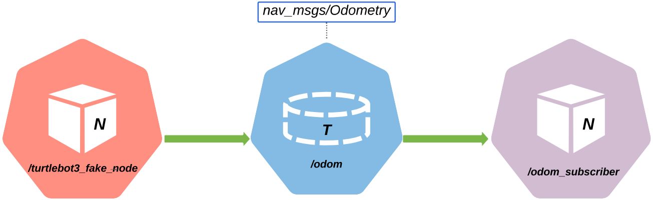
# Package | Publisher | Exercise#3

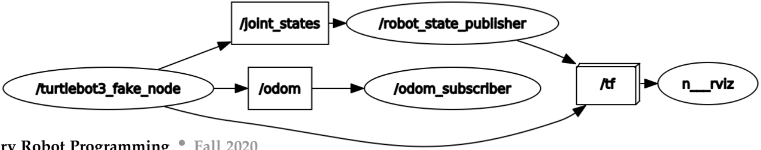
- Edit vel_publisher.launch to set the parameter publisher_rate on the parameter server through the launch file argument publisher_rate (<arg> tag).
- Edit vel_pub.cpp to read the rate value from the parameter server.

```
//--previous version
ros::Rate loop_rate(2);
//--new version
ros::Rate loop_rate(my_rate);
```

# Package | Subscriber

- Publishing Messages is only half of the story when it comes to communicating with other Nodes via Messages.
- The Node turtlebot3_fake_node publishes the pose of the robot on the Topic /odom.
- We want to write a Subscriber which will read the pose of the robot from the Topic /odom and outputs it in the terminal.





# Package | Subscriber

- The nav_msgs/Odometry message stores an estimate of the position and velocity of a robot in free space.
- In our current application, this message is published on the /odom topic.

```
roslaunch turtlebot3_fake turtlebot3_fake.launch
```

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

`rostopic echo /odom` (to keep listening to the topic)

or

`rostopic echo /odom -n 2` (to print 2 messages and exit)

# Package | Subscriber

- To retrieve Messages from the Topic /odom we need to know about the structure of those Messages. The structure will allow us to unpack the Messages in our Subscriber.

```
rostopic info /odom
```

```
Type: nav_msgs/Odometry

Publishers:
 * /turtlebot3_fake_node (http://linux:44759/)

Subscribers:
 * /rviz (http://linux:45243/)
```

# Package

- To see the structure of the message: `rosmsg show nav_msgs/Odometry`

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```

# Package | Subscriber

- Steps to create a Subscriber:
  1. Create a new package odom_subscriber in your workspace. This package has two dependencies: roscpp and nav_msgs
  2. In *odom_subscriber/src*, create a new file odom_sub.cpp (this file will contain code for our publisher node).
     - You can download this file from Canvas.
  3. Edit CMakeLists.txt and package.xml accordingly.
  4. Compile your package with `catkin build` and run your Node with `rosrun`
- Use slides 16–19 as examples to complete steps 1., 3., and 4.
- See next slide for step 2.

## Package | Subscriber

2. In *~/catkin_ws/src/odom_subscriber/src*, create a new file odom_sub.cpp
   - Steps for subscribing:
     (i) Initialize the ROS system.
     (ii) Subscribe to the /odom Topic.
     (iii) Spin, waiting for messages to arrive.
     (iv) When a message arrives, the OdomCallback() function is called.
- Full C++ tutorial on publisher/subscriber can be found here.

## Package | Subscriber

- One important difference between publishing and subscribing is that a Subscriber does not know when Messages will arrive on a Topic.
- To deal with this fact, we must place any code that responds to incoming Messages inside a callback function, which ROS calls once for each arriving Message.
- In computer programming, a callback is any executable code that is passed as an argument to other code; that other code is expected to call back (execute) the argument at a given time.
- A callback is a function that you do not call yourself, but define it yourself. Usually you pass the function pointer to another component that will call your function when it seems appropriate.

## Package | Publisher/Subscriber | Exercise#4

- In our catkin workspace we have separated the Publisher from the Subscriber by creating them in different packages. It is common to have multiple nodes in the same package.
  1. Create a catkin package publisher_subscriber. This package has 3 dependencies: roscpp, geometry_msgs, and nav_msgs
  2. Copy vel_pub.cpp from */catkin_ws/src/vel_publisher/src* and paste it in */catkin_ws/src/publisher_subscriber/src*
  3. Copy odom_sub.cpp from */catkin_ws/src/odom_subscriber/src* and paste it in */catkin_ws/src/publisher_subscriber/src*
  4. Edit CMakeLists.txt and package.xml accordingly.
  5. Run the Publisher and the Subscriber in different terminals.

## Next Class | 12/03

- Lecture 13 – Templates and Exception Handling.
- Quiz on ROS (Lectures 11 and 12) between 7:05-7:20 pm.