Introductory Robot Programming

ENPM809Y

Lecture 11 – The Robot Operating System (ROS) – Part I

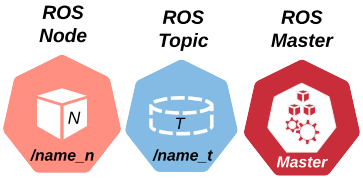
Zeid Kootbally
zeidk@umd.edu

Fall 2020

Overview I

Conventions

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



Highlights

- This lecture is the first of two lectures we have on ROS core components.
- Next lecture will show you how to write Subscribers and Publishers to read sensor data and to control robot actuators, respectively.
- Keywords: Node, Topic, Publisher, Subscriber, ROS Master.

What is Shell, Bash, and Terminal?

- Shell: A program which processes commands and returns output.
- GNU Bash or simply Bash: A Unix shell and command language written by Brian Fox for the GNU Project as a free software replacement for the Bourne shell.
  - A Bash script is a plain text file which contains a series of Bash commands.
  - Note: Anything you can run normally on the command line can be put into a script and vice-versa.
- Terminal: A text window program that runs a shell (Bash shell in Ubuntu). Shell commands will be displayed like `this` and need to be entered in a terminal.
  - Sometimes you will need to open multiple terminals to run a program. This is because some Bash commands are blocking and the only way to terminate these commands is to click in the terminal and then Ctrl-c
  - You can open a terminal with Ctrl-Alt-t
  - The Tab key is your friend when working with command lines.

The Robot Operating System (ROS)

- Online comments from Brian Gerkey[1]

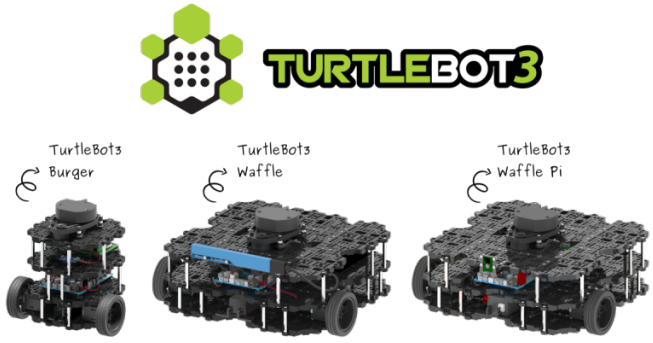
[1] https://answers.ros.org/question/12230/what-is-ros-exactly-middleware-framework-operating-system/

ROS | Resources

- ROS official website.
- ROS tutorials.
- ROS package indexes.
- ROS subreddit.
- ROS Discourse.
- ROS Answers.
- Get involved.
- Catkin Command Line Tools.

ROS | Robots Used in Lectures 11 and 12

- In this lecture and in the next one we will use the turtlebot3 mobile robots to learn about ROS concepts.
- Read about the origin of Turtle Robots.



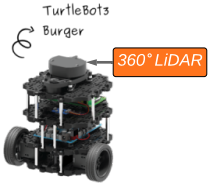
ROS | Node

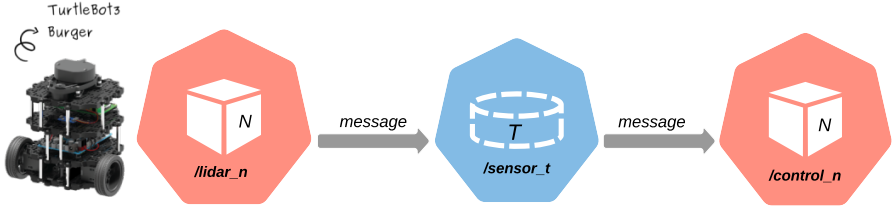
Node:

The core entity in ROS which is a small program written in Python or C++ within a ROS package that executes some relatively simple task or process.

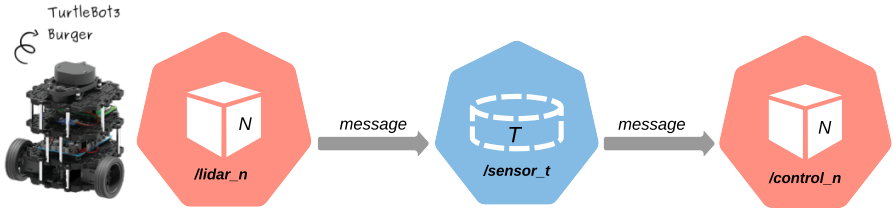
ROS | Node

- You can think of one Node (/lidar_n) which reads sensor data and another Node (/control_n) which controls a robot actuators based on the data received from /lidar_n.
- /lidar_n can communicate the sensor data to any other Node by publishing a Message to a Topic (/sensor_t).
- /control_n can retrieve that Message by subscribing to /sensor_t.





ROS | Node



- Nodes communicate with each other by publishing Messages to and reading Messages from Topics.
- Question: What is a Message?
- Answer: A packet of data (a simple data structure with typed fields). We will learn more about Messages in next lecture.
- Question: What is a Topic?
- Answer: Let's find out.

ROS | Topic

Topic:
A named bus[2] over which nodes exchange messages.

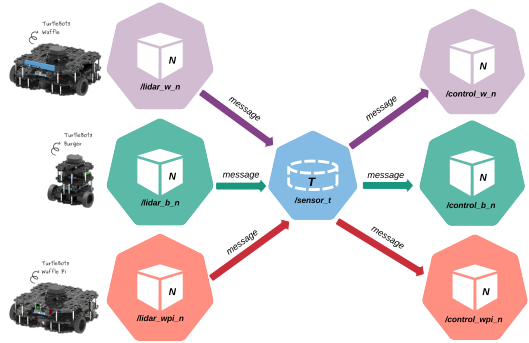
[2] A communication system that transfers data between components inside a computer, or between computers

ROS | Topic

- Each Topic is strongly typed by the Message type used to publish to it and Nodes can only receive Messages with a matching type.
- Topics have anonymous publish/subscribe semantics.
- In general, Nodes are not aware of who they are communicating with.
  - Nodes that are interested in data subscribe to the relevant Topic. These Nodes are called subscriber Nodes or Subscribers.
  - Nodes that generate data publish to the relevant topic. These Nodes are called publisher Nodes or Publishers.

ROS | Topic

- There can be multiple Publishers and Subscribers to a Topic.
- A Topic may be introduced and various Publishers may take turns publishing to that Topic.
- A Subscriber only needs to know the name of a Topic and does not need to know what Node or Nodes publish to that Topic.



ROS | ROS Master

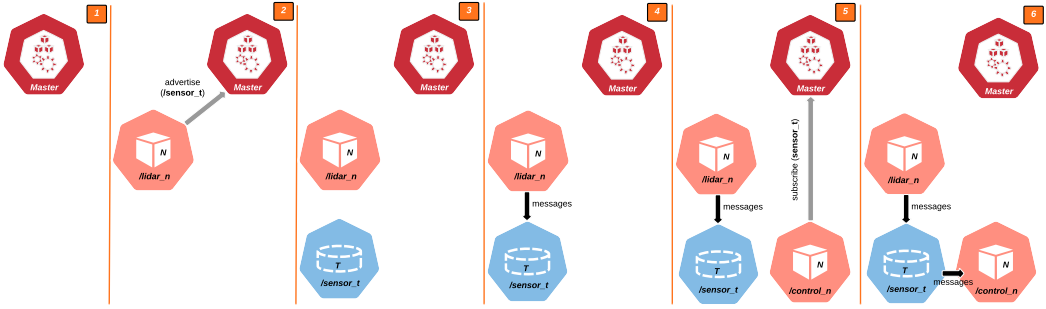
- Publishers and Subscribers both need to know how to communicate via a Topic.
  - This is accomplished with `roscore`, a collection of Nodes and programs that are pre-requisites of a ROS-based system.
- `roscore` starts a *ROS Master*, which is responsible for coordinating communications between Nodes and Topics, like an operator.
- Although there can be many ROS Nodes distributed across multiple networked computers, there can be only one instance of `roscore` running.
- The machine on which `roscore` runs establishes the master computer of the system.

ROS | ROS Master

- `roscore` must be running before any ROS Node is launched.
- To run `roscore`, open a terminal and enter `roscore`
  - The response to this command will be a confirmation `started core services`
- The terminal running `roscore` will be dedicated to `roscore`, and it will be unavailable for other tasks.
- `roscore` should continue running as long as the robot is actively controlled (or as long as desired to access the robot's sensors).
- To safely terminate a running `roscore`, click in the terminal and press Ctrl-c

ROS | ROS Master | Publisher First

- After `roscore` has been launched (which starts the ROS Master), a Publisher may start publishing to a Topic and a Subscriber may subscribe to the Topic to start receiving Messages.
- Note: A Publisher may initiate a Topic if the Topic does not exist.

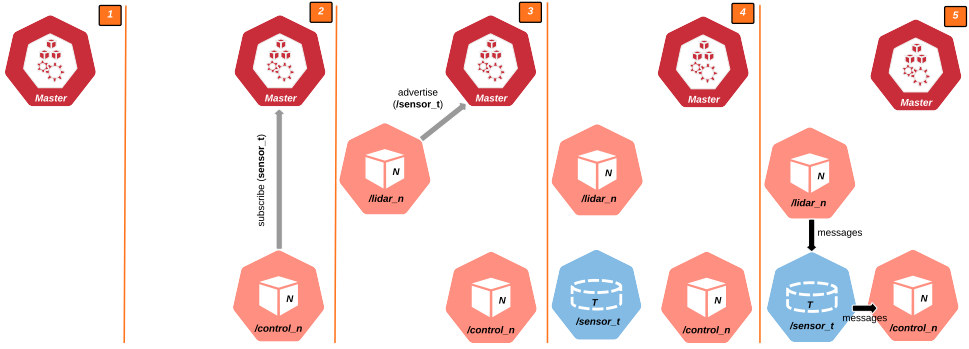


ROS | ROS Master | Subscriber First

- Nodes can be started and stopped independently of one another. This provides the flexibility to launch Publishers and Subscribers in any order to ease system design.
- ROS will allow the Subscriber to register its desire to receive Messages from a named Topic, even though that Topic does not exist.
- At some point, if or when a Publisher informs ROS Master that it will publish to that named Topic, the Subscriber's request will be honored, and the Subscriber will start receiving the published Messages.

ROS | ROS Master | Subscriber First

- The figure below shows the behaviors when a Node subscribes to a non existent Topic.



ROS | ROS Master

- In next lecture we will write our own Publishers and Subscribers to send commands to a robot and to read sensor data, respectively.
- We need to store this code in a *catkin package* (ROS package) which has a specific hierarchy within a *catkin workspace* (workspace that ROS can understand).
- We need to understand the following concepts:
  - What a catkin workspace is.
  - What a ROS package is.

ROS | Catkin Workspace

- A catkin workspace is a directory in which you can create or modify existing catkin packages.
- The catkin structure simplifies the build and installation process for your ROS packages.
- We will use packages which come with your ROS installation and we will create our own package in the catkin workspace.
- We need to do the following steps to create a catkin workspace:
  1. Add environment variables to your path.
  2. Create the catkin workspace.
  3. Initialize and build the catkin workspace.
  4. Overlay your catkin workspace on top of your environment.

ROS | Catkin Workspace

1. Add environment variables to your path to allow ROS to function:

```
source /opt/ros/melodic/setup.bash
```

or

```
. /opt/ros/melodic/setup.bash
```

   - When you `source` something in the terminal, each line in the "sourced" file (setup.bash in this case) is executed as is it were typed into the current terminal.
   - You can add the command "`source /opt/ros/melodic/setup.bash`" (without the quotes) in your .bashrc file, so that it will be executed every time you open a new terminal.

ROS | Catkin Workspace

2. Create your catkin workspace: `mkdir -p ~/catkin_ws/src`
   - Note: If you already have a directory named *catkin_ws*, choose another name for the new catkin workspace (or copy your current *catkin_ws* somewhere else or rename it).
   - `mkdir`: Command to create a directory.
   - `mkdir -p`: Command to create a directory and subdirectories.
   - `~`: Your Home directory.
   - `mkdir -p ~/catkin_ws/src` : Create the *src* directory inside the *catkin_ws* directory inside your *Home* directory.

ROS | Catkin Workspace

3.a Initialize the catkin workspace with `catkin config --init` (see here)

- `cd ~/catkin_ws`
- `catkin config --init`

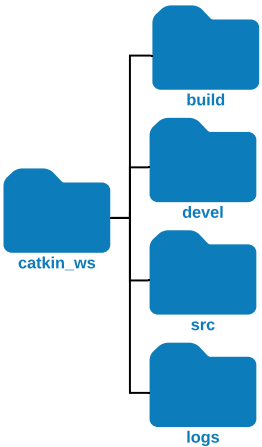
3.b Build the catkin workspace with `catkin build`

- You should execute this command in the root of your catkin workspace if `catkin config --init` was not previously executed. Otherwise, you can run it from anywhere in your workspace.
  - `cd ~/catkin_ws`
  - `catkin build`

ROS | Catkin Workspace

- Main differences between `catkin build` and `catkin_make` can be found here.
- Some advantages of `catkin build` over `catkin_make`
  - Ability to build the workspace from wherever in the directory (so you can build even if you are not in the root of the workspace).
  - Easy to build single packages and blacklist others.
  - Easy to store cmake arguments so that you don't have to repeat them every time.

ROS | Catkin Workspace



- *src*: This is where you can clone, create, and edit packages you want to build.
- *build*: This directory contains the intermediate build products for each package. Do not modify this folder.
- *devel*: This directory contains the final build products which can be used to run code, but relies on the presence of the source space. Do not modify this folder.
- *logs*: This directory contains logs from building and cleaning packages. Do not modify this folder.
- For more information, see Workspace Mechanics.

ROS | Catkin Workspace

4. Overlay your catkin workspace on top of your environment.
   - You should always source your local workspace setup last.
     1. `source /opt/ros/melodic/setup.bash`
     2. `source ~/catkin_ws/devel/setup.bash`
   - You should add this line in your .bashrc file.

ROS | Catkin Workspace | Environment Variables

- ROS uses a set of Linux system environment variables in order to work properly. You can check these variables by typing: `export | grep ROS`

```
declare -x ROSLISP_PACKAGE_DIRECTORIES=""
declare -x ROS_DISTRO="melodic"
declare -x ROS_ETC_DIR="/opt/ros/melodic/etc/ros"
declare -x ROS_MASTER_URI="http://localhost:11311"
declare -x ROS_PACKAGE_PATH="/home/zeid/catkin_ws/src:/opt/ros/melodic/share"
declare -x ROS_PYTHON_VERSION="2"
declare -x ROS_ROOT="/opt/ros/melodic/share/ros"
declare -x ROS_VERSION="1"
```

ROS | Catkin Workspace | Environment Variables

- The most important ones are:
  - `ROS_MASTER_URI` – Contains the url where the `roscore` is executed. Usually, your own computer (localhost).
  - `ROS_PACKAGE_PATH` – Contains the paths on your Hard Drive where ROS packages are located.
    - Make sure that your catkin workspace is listed here.

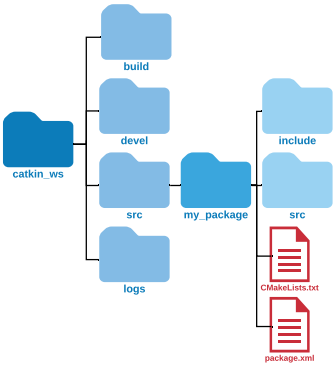
ROS | Package

- ROS uses packages to organize its programs. You can think of a package as all the files that a specific ROS program contains; all its C++ files, Python files, configuration files, compilation files, launch files, etc.
- For a package to be considered a catkin package it must meet a few requirements:
  1. The package must contain a catkin compliant package.xml file. That package.xml file provides meta information about the package.
  2. The package must contain a CMakeLists.txt which uses catkin to build the package (generate executables, install files, etc).
  3. Each package must have its own folder. This means no nested packages nor multiple packages sharing the same directory.

ROS | Package

- To create a package my_package which depends on the package roscpp:
  - `cd ~/catkin_ws/src`
  - `catkin create pkg my_package --catkin-deps roscpp`
    or
  - `catkin create pkg --catkin-deps roscpp -- my_package`
- Build your package:
  - `catkin build` to build all packages in your catkin workspace.
    or
  - `catkin build my_package` to build a specific package.
- More information on creating packages can be found here.

ROS | Package



- Files in a package (e.g., my_package) are organized with the following structure:
  - *src*: Directory for source files (.cpp).
  - *include*: Directory for source files (.h).
  - CMakeLists.txt: List of cmake rules for compilation.
  - package.xml (manifest): Package information and dependencies.
- Optional directories:
  - *launch* (optional): Directory for storing launch files.
  - *nodes* (optional): Directory for storing source files for nodes.
  - *script* (optional): Directory for storing scripts (e.g., Python scripts).
  - *msg* (optional): Directory for custom messages.
  - *srv* (optional): Directory for custom services.

ROS | Package | Package Navigation

- To go to any ROS package use `roscd <package_name>` [3]
- Entering only `roscd` in a terminal will take you to your catkin workspace.
- Try:

```
source /opt/ros/melodic/setup.bash
roscd
```

[3] http://wiki.ros.org/rosbash#roscd

ROS | Package | Package Search

- `rospack <command> [options] [package_name]` rospack is a command-line tool for retrieving information about ROS packages available on the filesystem. [4]
- Try `rospack help`
- We need some turtlebot3 packages for this section of the lecture. You can list all the packages that contain the word "turtlebot3".

```
rospack list | grep turtlebot3
```

- If nothing is found, you need to install the packages with:

```
sudo apt-get install ros-melodic-turtlebot3-*
```

[4] http://wiki.ros.org/rospack

ROS | Package | Packages not Found

- Sometimes ROS will not detect a new package when you have just cloned or created it.
- This is expressed by `roscd <package_name>` not working with your package.
- You can try 2 things to fix this issue:
  1. Make sure you did: `source ~/catkin_ws/devel/setup.bash`
  2. You can also force ROS to do a refresh of its package list with `rospack profile`

ROS | Package | Running Packages

- `roslaunch` is a tool to easily launch multiple ROS Nodes locally and remotely, to set parameters on the parameter server, etc.
- Syntax: `roslaunch <package_name> <launch_file>`
  - Launch the file `<launch_file>` that is located in the package `<package_name>`
- We would like to do the following:
  - Start the robot in an environment:
    ```
    roslaunch turtlebot3_fake turtlebot3_fake.launch
    ```
  - Drive the robot with key presses on the keyboard:
    ```
    roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
    ```

ROS | Package | Running Packages

- Running any of these 2 commands will raise the same error:

```
RLException:Invalid <arg> tag: environment variable 'TURTLEBOT3_MODEL' is not set.
```

- Let's take a look at one of these two launch files.
  - Go to the *launch* folder inside the turtlebot3_teleop package and check the turtlebot3_teleop_key.launch file (you can also open this file in a text editor).
    - `roscd turtlebot3_teleop/launch/`
    - `cat turtlebot3_teleop_key.launch`

ROS | Package | Running Packages

```
<launch>
  <arg name="model"
       default="$(env TURTLEBOT3_MODEL)"
       doc="model type [burger, waffle, waffle_pi]"/>

  <param name="model" value="$(arg model)"/>

  <!-- turtlebot3_teleop_key already has its own built in velocity smoother -->
  <node pkg="turtlebot3_teleop"
        type="turtlebot3_teleop_key"
        name="turtlebot3_teleop_keyboard"
        output="screen">
  </node>
</launch>
```

- All launch files are contained within a `<launch>` tag.

ROS | Package | Running Packages

```
<arg name="model"
       default="$(env TURTLEBOT3_MODEL)"
       doc="model type [burger, waffle, waffle_pi]"/>
```

- The `<arg>` tag allows you to create more re-usable and configurable launch files by specifying values that are passed via the command-line.
- In this example, if no value is provided for the argument `"model"` then a default value (stored in `TURTLEBOT3_MODEL`) is assigned to it.
- If `TURTLEBOT3_MODEL` has been defined and you pass the argument `"model"` to the command line then the value for `TURTLEBOT3_MODEL` is replaced with the value of the argument `"model"`
- In both cases you first need to set the environment variable `TURTLEBOT3_MODEL`
- Example:

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch model:=burger
```

ROS | Package | Running Packages

```
<param name="model" value="$(arg model)"/>
```

- The `<param>` tag defines a parameter to be set on the Parameter Server.
  - In this example, ROS will use the value for the argument `"model"` (in previous slide) to set the parameter `"model"` (in this slide) on the *Parameter Server*.

ROS | Package | Running Packages

- *Parameter Server*: A shared, multi-variable dictionary that is accessible via a network.
- Nodes use this server to store and retrieve parameters at runtime.
- The Parameter Server uses XMLRPC data types for parameter values, which include the following: 32-bit, Integers, Booleans, Strings, Doubles, ISO 8601 dates, Lists, and Base 64-encoded binary data.
- ROS has the `rosparam` tool to work with the Parameter Server.
- Some of the options include:
  - `rosparam list` – List all the parameters on the server.
  - `rosparam get <parameter>` – Get the value of a parameter.
  - `rosparam set <parameter> <value>` – Set the value of a parameter.
  - `rosparam delete <parameter>` – Delete a parameter.

ROS | Package | Running Packages

```
<node pkg="turtlebot3_teleop"
      type="turtlebot3_teleop_key"
      name="turtlebot3_teleop_keyboard"
      output="screen">
</node>
```

- The `<node>` tag specifies a ROS *node* that you wish to start.
- `pkg="package_name"` – Name of the package that contains the node you want to start.
- `type="executable_name"` – Name of the .cpp executable file or .py script that we want to execute.
- `name="node_name"` – Name of the ROS node that will launch our executable file.
- `output="output_channel"` – Through which channel you will print the output of the program.

ROS | Package | Running Packages

- The launch file located in the package turtlebot3_fake use the same argument `"model"` to set the robot model.
- The only difference is that a different name is used for the parameter on the *Parameter Server*.
  - Go to the *launch* folder inside the turtlebot3_fake package and check the turtlebot3_fake.launch file (you can also open this file in a text editor).
    - `roscd turtlebot3_fake/launch/`
    - `cat turtlebot3_fake.launch`

ROS | Package | Running Packages

```
<launch>
  <arg name="model"
       default="$(env TURTLEBOT3_MODEL)"
       doc="model type [burger, waffle, waffle_pi]"/>

  <param name="tb3_model" value="$(arg model)"/>
  ...
</launch>
```

- As we can see the content of the `<arg>` tag is exactly the same as the one described in turtlebot3_teleop_key.launch
- However, the parameter name used in this launch file (`"tb3_model"`) is different from the one used in turtlebot3_teleop_key.launch, which is `"model"`
- If you want to set the model of the robot on the Parameter Server you can use the parameter `"model"` as follows:

```
roslaunch turtlebot3_fake turtlebot3_fake.launch model:=burger
```

ROS | Package | Running Packages

- 2 ways to set the robot model.
  1. Set the value for the environment variable `TURTLEBOT3_MODEL` in each terminal: Not practical if the robot model needs to be set in multiple terminals.
  2. Set the value for the environment variable `TURTLEBOT3_MODEL` in .bashrc. This is better since everytime you start a terminal .bashrc is loaded, which in turns set the environment variable `TURTLEBOT3_MODEL`.

ROS | Package | Running Packages

1. `export TURTLEBOT3_MODEL=<name>` by replacing `<name>` with one of the three options: `burger`, `waffle`, or `waffle_pi`
   - `export TURTLEBOT3_MODEL=burger` or
   - `export TURTLEBOT3_MODEL=waffle` or
   - `export TURTLEBOT3_MODEL=waffle_pi`
   - Note: You are setting the value for this variable only in the current terminal. You will have to repeat this process in each terminal you open from now on.

ROS | Package | Running Packages

2. Store the robot model in your .bashrc file.
   - Open .bashrc with a text editor.
   - At the end of your file add one of the following:

     ```
     export TURTLEBOT3_MODEL=burger
     export TURTLEBOT3_MODEL=waffle
     export TURTLEBOT3_MODEL=waffle_pi
     ```

   - Save changes made to .bashrc
   - In the current terminal:

     ```
     source ~/.bashrc
     ```

   - Check `TURTLEBOT3_MODEL` is properly set:

     ```
     echo $TURTLEBOT3_MODEL
     ```

ROS | Package | Running Packages

- After setting a default value for `TURTLEBOT3_MODEL`, you can change it on the command line.

```
roslaunch turtlebot3_fake turtlebot3_fake.launch model:=burger
```

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch model:=burger
```

- or

```
roslaunch turtlebot3_fake turtlebot3_fake.launch model:=waffle
```

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch model:=waffle
```

- or

```
roslaunch turtlebot3_fake turtlebot3_fake.launch model:=waffle_pi
```

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch model:=waffle_pi
```

ROS | Package | Running Packages

- After setting the default value for the argument model we can launch our launch files.
- In the commands below we choose to modify the robot model with the use of argument.
  - Start RViz with a robot:

    ```
    roslaunch turtlebot3_fake turtlebot3_fake.launch model:=burger
    ```

  - Drive the robot with key presses on the keyboard:

    ```
    roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch model:=burger
    ```

- You can also check the values of parameters on the Parameter Server.

  ```
  rosparam list
  ```

  ```
  rosparam get model
  ```

  ```
  rosparam get tb3_model
  ```

ROS | Visualization

- RViz (ROS Visualization) is a powerful robot 3D visualization tool.
- It provides a convenient GUI to visualize sensor data, robot models, and environment maps, which are useful for developing and debugging your robot controllers.
- It is not a simulation environment (physics is not used in RViz).
- When working with ROS it is expected that you will need to use RViz very often for visualization and debugging.

ROS | Simulation

- Gazebo is a 3D simulation environment.
- Combining both ROS and Gazebo results in a powerful robot simulator.
- With Gazebo you are able to create a 3D scenario on your computer with robots, obstacles and many other objects.
- Gazebo also uses a physical engine for illumination, gravity, inertia, etc.
- You can evaluate and test your robot in difficult or dangerous scenarios without any harm to your robot.
- Originally, Gazebo was designed to evaluate algorithms for robots (error handling, battery life, localization, navigation and grasping).
- Gazebo was later improved to fulfill the need for a multi-robot simulator.
- You can see one of the default Gazebo worlds with:

```
roslaunch gazebo_ros willowgarage_world.launch
```

ROS | Simulation

- Turtlebot3 in Gazebo.
  - You can spawn the turtlebot in one of the existing worlds that come with Gazebo.
  - Note: In the commands below we are using the robot model stored in .bashrc
  - In the first terminal, run one of the following commands:
    - `roslaunch turtlebot3_gazebo turtlebot3_world.launch`
    - `roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch`
    - `roslaunch turtlebot3_gazebo turtlebot3_house.launch`
  - In a second terminal:
    - `roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch`
  - Move the robot around with the keys on your keyboard.
  - Ctrl-c to exit.

ROS | ROS Tools

- In the two following commands:

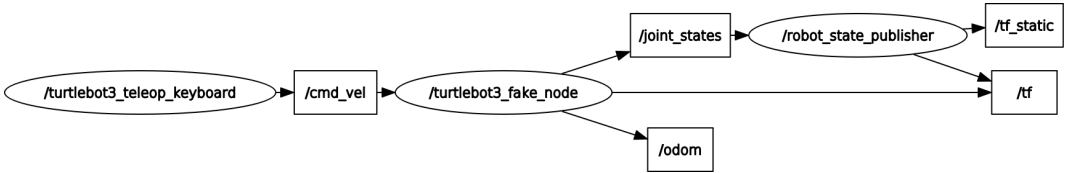
```
roslaunch turtlebot3_fake turtlebot3_fake.launch
```

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

  - Which package has a Publisher that publishes on the topic that accepts velocity messages?
  - Which package has a Subscriber that reads the topic that accepts velocity messages?
- If you don't know the answer to those questions you can use the `rqt_graph` command[5] by just running `rqt_graph`

[5] see http://wiki.ros.org/rqt_graph

ROS | ROS Tools



- Ovals represent nodes, rectangles represent topics, and the directed edges represent show publishers and subscribers.
- The node /turtlebot3_teleop_keyboard publishes messages on the topic named /cmd_vel
- The node /turtlebot3_fake_node is subscribed to the topic /cmd_vel

ROS | ROS Tools

- Another way to get information on ROS nodes is with the `rosnode` command[6].
- `rosnode list` displays all the active nodes.
- `rosnode info <node_name>` provides more information on the node `<node_name>`

[6] see http://wiki.ros.org/rosnode

ROS | ROS Tools

- `rosnode list`

```
/robot_state_publisher
/rosout
/rviz
/turtlebot3_fake_node
/turtlebot3_teleop_keyboard
```

ROS | ROS Tools

- `rosnode info /turtlebot3_fake_node`

```
Subscriptions:
 * /cmd_vel [geometry_msgs/Twist]
...
Connections:
...
 * topic: /cmd_vel
    * to: /turtlebot3_teleop_keyboard (http://robotagility:39973/)
    * direction: inbound
    * transport: TCPROS
```

ROS | ROS Tools

- Subscriber functions can be launched before the corresponding publisher functions.
- ROS will allow the subscriber to register its desire to receive messages from a named topic, even though that topic does not exist.
- At some point, if or when a publisher informs *ROS Master* that it will publish to that named topic, the subscriber's request will be honored, and the subscriber will start receiving the published messages.
- The flexibility to launch publisher and subscriber nodes in any order eases system design.

ROS | ROS Tools

- Terminate the `roslaunch` commands you started with Ctrl-c
- Restart these commands in reverse order:
- `roslaunch turtlebot3_fake turtlebot3_fake.launch`
- `rosnode info /turtlebot3_fake_node`
  - The node /turtlebot3_fake_node is subscribed to the topic /cmd_vel but the type of this topic is [unknown type].
  - Also note the absence of the topic /cmd_vel in the Connections section. You can also see this with `rqt_graph`
  - Explanation: /turtlebot3_fake_node registered its desire to receive messages from /cmd_vel even though this topic does not exist.

ROS | ROS Tools

- `roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch`
- `rosnode info /turtlebot3_fake_node`
  - In *Subscriptions, the type for the topic /cmd_vel is now [geometry_msgs/Twist].
  - The topic /cmd_vel now shows up in the Connections section.
  - /turtlebot3_teleop_keyboard is the node that initiates /cmd_vel along with the type of message this topic accepts.
  - Now /turtlebot3_fake_node can start receiving messages.

ROS | Summary

- 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?
  - We will see this in next lecture.

Next Class | 11/19

- Lecture 12 – The Robot Operating System (ROS) – Part II.
  - Make sure you understand the concepts discussed today before next week.
  - We will write C++ nodes (Publishers and Subscribers).
- No Quiz.
- No Assignment.