# Robot Operating System assistance sheet

Table of contents:

# 1. Basics of ROS:

- For detailed explanation of all the topics covered here visit: http://wiki.ros.org/
- For better understanding here are some slides: https://docs.google.com/presentation/d/1KBc8gXuB7iRfSZVZKymB-WWmuelxNX4I9XBWPFZidKA/edit?usp=sharing
- This document is written based on the official ROS WIKI and Udemy course about ROS by Anis Koubaa ( https://www.udemy.com/course/ros-essentials/ )
- 10 year montage of ROS: https://vimeo.com/245826128
- The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.
- Why? Because creating truly robust, general-purpose robot software is *hard*. From the robot's perspective, problems that seem trivial to humans often vary wildly between instances of tasks and environments. Dealing with these variations is so hard that no single individual, laboratory, or institution can hope to do it on their own.
- As a result, ROS was built from the ground up to encourage *collaborative* robotics software development. For example, one laboratory might have experts in mapping indoor environments, and could contribute a world-class system for producing maps. Another group might have experts at using maps to navigate, and yet another group might have discovered a computer vision approach that works well for recognizing small objects in clutter. ROS was designed specifically for groups like these to collaborate and build upon each other's work, as is described throughout this site.

## Requirements

- You have to have experience using **Ubuntu**. Personally, **I prefer Kubuntu**, which is much better than vanilla Ubuntu. ROS natively **supports Ubuntu and it's flavors**. So use these for best compatibility. Also, it's best practice **not to mix ROS distributions**.
- Python language
- Basics of C++ language ( for arduino interfacing )
- Patience and debugging skills.

# 2. Install tips:

- http://wiki.ros.org/ROS/Installation
- In the ROS website, all previous versions of ROS versions will be available but you should Install the LTS version that has matured for at least 2 years, so that all the packages are available.
- All the ROS versions also list out the compatible versions of Ubuntu as well. For example, Melodic supports Ubuntu 18.04 and all it's flavors, Kinetic supports 16.04.
- Use ROS wiki to install the full version.
- A Python/C++ editor will be needed, any program can be used but Visual studio code is recommended, as it is lighter.

# 3. Setup workspace & ROS package

- Workspace:
  - Install directory: opt/ros
  - Every Time terminal opens, to work on ROS, the setup.bash must be executed, thus in terminal:
    - **nano ~/.bashrc**
    - at the end of the file: **source /opt/ros/melodic/setup.bash**
  - Create a workspace to work on. To do that use catkin workspace
    - **mkdir -p ~/catkin_ws/src**
    - Go in the dir: **cd catkin_ws**
    - Compile the workspace: **catkin_make**
    - To enable the workspace every time, so in terminal:
      - **nano ~/.bashrc**
      - **at the end: source /home/hasib/catkin_ws/devel/setup.bash**
- Package:
  - Your projects on ROS are known as ROS Packages
    - To create a package,
      - go to the **src** of your workspace
      - in terminal: **catkin_create_pkg name_of_package dependencies_of_the_projects**
      - naming convention: **the package name must be all lower case**
      - common dependencies that **must be used**:
        - **std_msgs rospy roscpp**
      - example:
        - **catkin_create_pkg beginner_tutorials std_msgs rospy roscpp**

- the dependencies can be added later as well
- When using **rosrun** to run the packages, you can always press 'tab' twice after typing package name to know the nodes inside the package
- After creating a new package, compile it.
    - go to the workspace **root** folder
    - **catkin_make**
- in catkin_ws/src/ :
    - there's another 'src', all the code goes there
    - in the 'include' folder, all the libraries go
- In the visual studio, drag and drop the folder is possible and new files can be created.
- Everytime editing any file in 'src', the workspace will need to be recompiled
- Get a package from github: **git clone <link of git repo>**

# 4. ROS concept:

- To Start working on ROS, in terminal:
    - roscore
- To see all the ROS node:
    - rosnode list
- To see topics:
    - rostopic list

# 5. ROS topics, nodes, massages:

- First run, **roscore**
- rosNODE:
    - ROS node is an application that can be executed. Nodes can publish or subscribe to each other using ROS topics and send values to each other called ROS massages.
    - To run a ros application:
        - **rosrun <application name>**
            - **rosrun turtlesim** //'tab' 2 times shows all commands related
            - **rosrun turtlesim turtlesim_node** //turtlesim gui should come
    - Some ROS node commands:
        - **rosnode** //shows all commands
        - **rosnode list** //list of all nodes running
        - **rosnode info <name of node>** // give full info about a node

- Topic:
  - ROS topics are just mediums that is used by nodes to send ROS massages
  - Some ROS topic commands:
    - **rostopic** //shows all commands
    - **rostopic list** //shows all the topics currently running
    - **rostopic info <name of topic>** //shows info about the topic
    - **rostopic type** //show the type of the bus
    - **rostopic echo <name of topic>** //prints out the values being transferred
    - **rostopic pub <topic> <msg type> <args>** //directly sends msg to node
      - **rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'**
      - '-1' here tells the publisher to run once then exit
      - '--' tell it none of the ags are an option, required when using negative number
  - Using rqt graph:
    - Install:
      - **$ sudo apt-get install ros-<distro>-rqt**
      - **$ sudo apt-get install ros-<distro>-rqt-common-plugins**
    - Commands:
      - **rosrun rqt_graph rqt_graph** // visually shows connected nodes and topics
      - **rosrun rqt_plot rqt_plot** //shows plots of the topics over time

- Messages:
  - ROS nodes generate values that get sent via topics to other nodes, the values are called messages.
  - Some ROS msg commands:
    - **rosmsg** //show all commands
    - **rosmsg show <name of a topic type>** //shows the structure of values that are sent

# 6. How ROS works:

- Basic info:
  - ROS is a collection of nodes
  - Nodes that give massages are **publishers**
  - Nodes that take massages are **subscribers**
  - The nodes exchange messages between each other using certain **Topics**
  - The messages will be of a certain **format/type.**

- How publisher and subscriber communicate:
  - **Master node** is responsible for identifying all the nodes, topics and messages and communicates which nodes are publishers and subscribers and what topic are they to use.
  - So, lets say:
    - There's node1, which connects with roscore and gives its name, subscribing node name , subscribing topic name, required topic and massage type.
    - roscore will store this information.
    - Another node, node2 now connects to the roscore and also gives its name, publishing topic name, message type.
    - Since roscore has global knowledge, it will tell node2 that node1 has it's required topic name, massage type and is a publisher.
    - Node2 will connect to node1 using the rosTCP protocol.
    - Node1 replies back to node2 using the rosTCP protocol.
    - The publishing node(node1) sends messages via topics periodically, which is coded into the publishing node.
    - The subscribing node(node2) will have a **callback function** that executes whatever message the publisher sends.

  - Publisher node creation rules:
    - determine **NAME of topic** to publish.
    - determine the **type of massage** to publish.
    - determine the **frequency of topic publication**. How many massages per sec.
    - Create a **publisher object** with parameters chosen
    - **Keep publishing the topic** message using while loop at the selected frequency

- Subscriber node creation rules:
  - Identify the **name for the topic** to listen to. This allows which channel to use for communication.
  - Identify the **type of messages** will be received. This allows how to extract the information.
  - Define **a callback function** that will automatically be executed when a new message arrives.
  - **Start listening** for the topic messages

# 7. Writing publisher and subscriber nodes:

- Nodes can be written in C++ or Python. I prefer python for the moment.
- for detailed info on the codes below, visit: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

## Publisher node:

- **#!/usr/bin/env python** #mandatory,this line declare the program as python

- **import rospy** #import library required for ros
- **from std_msgs.msg import String** '''the type of message, if the message is custom, then from pacakage_name.msg import type'''

- **def talker():**
  - **rospy.init_node("Talker", anonymous = True)** '''very important, as this declares node name, anonymous = True enables the nodes to have a unique name when executed, if the same name node is launched, the previous node is kicked off. Thus, anonymous = true gives the power to run multiple versions of the same node.'''

  - **pub = rospy.Publisher("chatter", String, queue_size=10)**

'''declaring the node as a publisher, "chatter" is the topic name, String is the message type and queue_size=10 is the maximum queued messages if . . subscriber fail to receive them'''

.

  - **rospy.rate(10)** '''The rate of which the node sends messages, 10 means 10 messages per second, the nodes have to process the code at that speed as well, thus be careful of this setting'''

  - **while not rospy.is_shutdown():** #checking the node is still running
    - **hello_str = 'This is a message'**
    - **rospy.loginfo(hello_str)**

'''This prints msg on the screen, logs activity and writes it to rosout, . rosout is used to debug'''

- **rospy.publish(hello_str)** **#**publishes the message
- **rate.sleep()** **#**gives a chance to exit the node

- **if __name__ = '__main__':**
  **try:**
  **talker()**
  **except:**
  **rospy.ROSInteruptException** **#**prevents accidental executing . . even after shutting the node down
  **pass**

- Subscriber node:

- **#!/usr/bin/env python**
- **import rospy**
- **from std_msgs.msg import String**
- **def callback(message):**
  - **rospy.loginfo("I heard %s", message.data)**
- **def listener():**
  - **rospy.init_node("Listener", anonymous = True)**
  - **rospy.Subscriber("chatter", String, callback)** '''make the node a subscriber, "chatter" is the topic it'll want, String is the type of message and callback is a function that executes the commands received from the publisher.'''
  - **rospy.spin()** **#**Keep the code running until it's shutdown
- if __name__ = '__main__':
  - listener()

- Some troubleshooting while writing nodes
  - Make sure to name packages properly and navigate to them properly
  - when writing the topic to a node if writing for an existing node, like turtlesim, make sure to write the whole topic name.
    - rospy.Publisher("turtle1/cmd_vel", Twist……)
  - While writing subscriber node, make sure that you use the data passed in the function
    - def callback(pose_msg):
      - rospy.loginfo("%s", pose_msg.x)

# 8. Create custom ROS messages:

- ROS does not include all types of messages, thus when a new device with a different type of messages is used, we have to create a new message type in ROS.
- ROS message structure:
    - lets understand the structure of a ROS message.
        - package_name/message_type
            - **std_msgs/String**
            - **geometry_msgs/Twist**
- ROS message content:
    - Each message type contains the type and a field that contains that kind of data
        - type field
            - **string data** //here, String is the type and data contains the values
            - data types can be found here:
                - http://wiki.ros.org/msg

- Steps to create a new ROS message:
    - create a **msg** folder in your **package, not anywhere else**
    - create the message file with the extension **.msg**
    - Edit the .msg file by adding attributes and one line per attributes.
        - ie: **int32 id** // here int32 is the type and id is the field will contain data
        - data types can be found here:
            - http://wiki.ros.org/msg
    - Update the dependencies:
        - in **package.xml**
            - make sure these files are there:
                - **<build_depend>message_generation</build_depend>**
                - **<exec_depend>message_runtime</exec_depend>**
        - in **CMakeList.txt**
            - **Add** these lines in the following **functions,** make sure to **uncomment** the **#** commented functions**.**
                - **find_package**:
                    - **messsage_generation**
                - **add_message_files**
                    - edit **message1.msg** to your **.msg** name, ie: **IOTSensor.msg**
                - **generate_message**
                    - **uncomment**
                - **catkin_package**

        - in the line, “CATKIN_DEPENDS roscpp rospy std_msgs” add **message_runtime**

    - Compile the package using **catkin_make**
    - To check if the message is created, use **rosmsg show**
- Some troubleshooting:
    - keep a **backup** of Cmake file
    - double check the edits
    - **in .msg** file make sure you write the **types** properly. ie:
        - **String name** //wrong, S is capital, it should be **string** or it'll give **‘massage_generation”** error while **catkin_make** is compiling**.**
    - if the message is custom, then **from pacakage_name.msg import type**
        - **ie: from iot_bot.msg import iotSensor**

# 9. ROS Services:

- ROS services work on a one time communication, there are 2 parts, ROS Service and ROS Client. Client requests for a service, the server serves the service.
- ROS services are used when a specific action is needed to be performed.
    - Spawning a new turtle in turtlesim
- Commands:
    - **rosservice**
    - **rosservice list** //gives the list of all servies of all nodes running
    - **rosservice info <name of service>**
        - gives the node of the service
        - location of the node
        - type of the service
        - the arguments that it processes
    - **rossrv info turtlesim/spawn //**gives the parameters that the service type gives
    - **rosservice call /spam 7 7 0 t2**
        - executes the service
        - /spam 7 7 0 t2 //name of service x_val y_val theta_val name
    - CMD tools to check the services
        - **rossrv list //**all the ros services in the system
        - **rossrv show <name of service>**

- Creating a new ROS service:
    - Define the service message, define the request response data type.
        - Go to you package folder and create a folder named **srv**, just like msg
        - Inside the **srv** folder, create a file with the extension **.srv,** just like msg
        - in the **.srv** file, add the **type data**

        - since service has 2 parts, server and client, the .srv file will also have 2 parts, ie:
            - **int64 a** //the client sent a and b's values using the service
            - **int64 b**
            - **---** //divider
            - **int64 sum** **//**the server responds with the **sum** of a, b
    - Compile the **srv** file
        - **package.xml**
            - **<build_depend>message_generation</build_depend>**
            - **<exec_depend>message_runtime</exec_depend>**
        - **cmakelist.txt** (python)
            - add the lines in the **functions**:
                - **find_package**
                    - **message_generation**
                - **add_service_files**
                    - your **.srv** file name
    - compile workspace
        - **catkin_make**
    - verify
        - go to catkin_ws>devel>include>packagename, the .srv file should be ther

- Create ROS node that has the service implemented
- create ROS node that can be the client for the service
- Execute the service
- Consume the service by the client

-

- Writing ros service nodes
  - **detailed:** http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29
  - **Server:**
    - **Steps:** the server can be implemented in a Publisher/subscriber node
      - import everything from package.srv
      - create function
        - create ros node
        - **rospy.Service("name of service", service_type, handler_function)** //much like subscriber node, handler_fucntion executes any commands that the service need to perform
        - loginfo
        - **rospy.spin**
      - create handler function:
        - to return the result **name_of_serviceResponse(..)**
  - **Client:**
    - **steps:** client sends the command, it will appear in services list
      - take input from user using **argv** list in python
      - pass the **argv** list values in a function that will do the rest
      - create a function that takes the **argv** values
        - wait for the service to be available
          - **rospy.wait_for_service(**"name of service")
        - create the service proxy that'll pass the arguments to the server
          - **service = rospy.ServiceProxy("name of service", type of service)**
        - use the **service** object to pass the arguments
          - **req = service(arg1, ag2,...)**
          - **req** receives the return value sent by the server
        - return the server message that was received by **req**
          - **return req.name_of_server_variable_in_serv_file**

# 10. Network configuration:

When running ROS on different devices on the same network, the following settings need to be set:

- Config the robot
  - On the robot ROS, the **roscore** will be running
  - in terminal:
    - **nano ~/.bashrc,** add the following lines (copy paste)
      - **#Robot machine Configuration**
      - **#the localhost IP address - IP address for the master node**
      - **export ROS_MASTER_URI=**http://localhost:11311
      - **#this is telling other ROS nodes the location of the master #node, 11311 is the default port**
      - **#The IP address for the Master node**
      - **export ROS_HOSTNAME=** <enter ip address of the robot machine, Find ip address using **ifconfig** in terminal 192.168.1.2>
      - **export ROS_IP=**<enter ip address of the robot machine 192.168.1.2>
      - **#lines above send the ip address of the master node**
      - **#printing info**
      - **echo ”ROS_HOSTNAME: “ $ROS_HOSTNAME**
      - **echo “ROS_IP: “$ROS_IP**
      - **echo “ROS_MASTER_URI: “$ROS_MASTER_URI**

- Config the devices in the network:
  - **do not run roscore on this device after the config**
  - In terminal: (copy paste)
    - **nano ~/.bashrc**
    - **#network device config**
    - **export ROS_IP=**<enter device’s ip address ie: 192.168.1.2>
    - **export ROS_HOSTNAME=**<enter device’s ip address ie: 192.168.1.2(same as ros_ip)>
    - **export ROS_MASTER_URI=**<enter robot’s ip address with port, or where the roscore will run> **(ie: http://192.168.2.1:11311)**
    - **#print info**
    - **echo “ROS_HOSTNAME: “$ROS_HOSTNAME**
    - **echo “ROS_IP: “$ROS_IP**

        - **echo “ROS_MASTER_URI: “$ROS_MASTER_URI**

# 11. ROS Launch command:

- To run multiple nodes and give parameters at once, **roslaunch** is used
- The files are written in XML format
- Placed in **launch** folder in the package folder, or can be placed in subfolder of the package folder
- **Create roslaunch file:**
    - **<launch>**
        - **<node pkg=”name_of_package1” type=”name_of_node1” name=”give name” />**
        - **<node pkg=”name_of_package2” type=”name_of_node2” name=”give name” />**
    - **</launch>**
- Use other roslaunch files in a new roslaunch file
    - **<launch>**
        - **<include file=”$(find package_name)/src/subfolder_if_exsists/existing_launch_file.launch**
        - **<node pkg=”name_of_package1” type=”name_of_node1” ”/> name=”give name” />**
        - **<node pkg=”name_of_package2” type=”name_of_node2” name=”give name” />**
    - **</launch>**

# 12. Using ROS serial:

- For more details: [http://wiki.ros.org/rosserial](http://wiki.ros.org/rosserial)
    - rosserial protocol allows you to connect with devices for which drivers are not yet written in ROS, usually microcontrollers like Arduino.
    - How to run
        - **rosrun rosserial_python serial_node.py <port of arduino>**
            - **run this on the device arduino is connected to.**
        - **rostopic list**
            - there should be a **chatter** topic now
        - **rostopic echo /chatter**
            - **can be run on other devices.**
    - ROS serial architecture:
        - **rosserial_python** runs on device
        - **ros message**
        - **rosserial_ardino runs** on the Arudino
            - arduino can be set up into Subscriber or Publisher

- **Installing rosserial:**
  - `sudo apt install ros-melodic-rosserial-arduino`
  - Install arduino IDE from the website
  - Generate ROS library folder in Arduino libraries folder
    - **rosrun rosserial_arduino make_libraries.py .**
- **Installing rosserial on ros melodic Raspberry pi4**
  - **https://www.intorobotics.com/how-to-install-ros-melodic-rosserial-and-more-on-raspberry-pi-4-raspbian-buster/**
  - **$cd catkin_ws/src**
  - **https://github.com/ros/common_msgs**
  - **$cd ..**
  - **$catkin_make**
  - **$catkin_install**
  - **$cd src**
  - **$git clone https://github.com/ros-drivers/rosserial.git**
  - **$cd ..**
  - **$catkin_make**
  - **$catkin_install**
  - tip; install ros_lib in the arduino ide from the library manager, version 7.9

## ○ Creating rosserial arduino pub/sub:

- In the examples, many ros arduino codes ar given
- Ros Arduino uses C++ language
- Running the arduino code in rosserial
  - **rosrun rosserial_python serial_node.py <port number>**
- **General structure of a arduino publier using c++**
  - **#include <ros.h>**
  - **#include <std_msgs/String.h>** //message type
  - **std_msgs : : String str_msg;** //creating message
  - **ros : : NodeHandle node;** //creating instance of node
  - **ros : : Publisher pub_var("typic_name", &str_msg);**
  - **void setup(){**
    - **node.initNode();** //node initiation
    - **node.advertise(chatter);** //publishing the node
  - **}**
  - 
  - **void loop(){**
    - **str_msg.data = "hello world";**
    - **chatter.publish(&str_msg);**
    - **node.spinOnce();**
  - **}**

○ **Creating ros arduino subscriber:(blink)**

- to run the code: **rostopic pub toggle_led std_msgs/Empty --once**
- **#include <ros.h>**
- **#include <std_msgs/Empty.h>**
- **ros:: NodeHandle node;**
- **void callback(const std_msgs::Empty& toggle_msgs){**
    - **digitalWrite(13, HIGH);**
    - **delay(1000);**
    - **digitalWrite(13, LOW);**
- **ros:: Subscriber<std_msgs::Empty> sub(“toggle”, &callback);**
- **void setup(){**
    - **pinMode(13, OUTPUT)**
    - **node.initNode()**
    - **node.subscribe(sub);**
- **}**
- **void loop(){**
    - **node.spinOnce();**
- **}**

# 13. Gazebo