جَامعة
المَنارة
MANARA UNIVERSITY

Introduction to Robot Operating System (ROS 1)

Playing with ROS nodes, topics and messages- turtlesim example

discusses the use of: roscore, rosnode, and rosrun commandline tools

Dr. Essa Alghannam

ROS Messages

- Messages: Data structures used for communication. Nodes communicate with each other through messages that are published by topics.
- a simple structure of a message contains specific fields representing the data type being shared. (standard types or types developed by the user).
- ROS has a lot of messages predefined, but if you develop a new message, it will be in the msg/ folder of your package.
- A message must have two principal parts: fields and constants.

1- Fields define the type of data to be transmitted in the message.

2- Constants define the name of the fields.

[field] [constant]

Int32 number

ROS has the command-line tool rosmsg to get information about messages.

- rosmsg show msg_name or rosmsg info msg_name: Show message description (displays the fields of a message.)

```
essa@essa:~$ rosmsg show Int32
[std_msgs/Int32]:
int32 data
```

- rosmsg package topic_name List messages in a package
- rosmsg packages msg_name List packages that contain messages

rosmsg package std_msgs

std_msgs/Bool
std_msgs/Byte
std_msgs/ByteMultiArray
std_msgs/Char
std_msgs/ColorRGBA
std_msgs/Duration
std_msgs/Empty
std_msgs/Float32
std_msgs/Float32MultiArray
std_msgs/Float64
std_msgs/Float64MultiArray
std_msgs/Header
std_msgs/Int16
std_msgs/Int16MultiArray
std_msgs/Int32
std_msgs/Int32MultiArray
std_msgs/Int64

std_msgs/Int64MultiArray
std_msgs/Int8
std_msgs/Int8MultiArray
std_msgs/MultiArrayDimension
std_msgs/MultiArrayLayout
std_msgs/String
std_msgs/Time
std_msgs/UInt16
std_msgs/UInt16MultiArray
std_msgs/UInt32
std_msgs/UInt32MultiArray
std_msgs/UInt64
std_msgs/UInt64MultiArray
std_msgs/UInt8
std_msgs/UInt8MultiArray

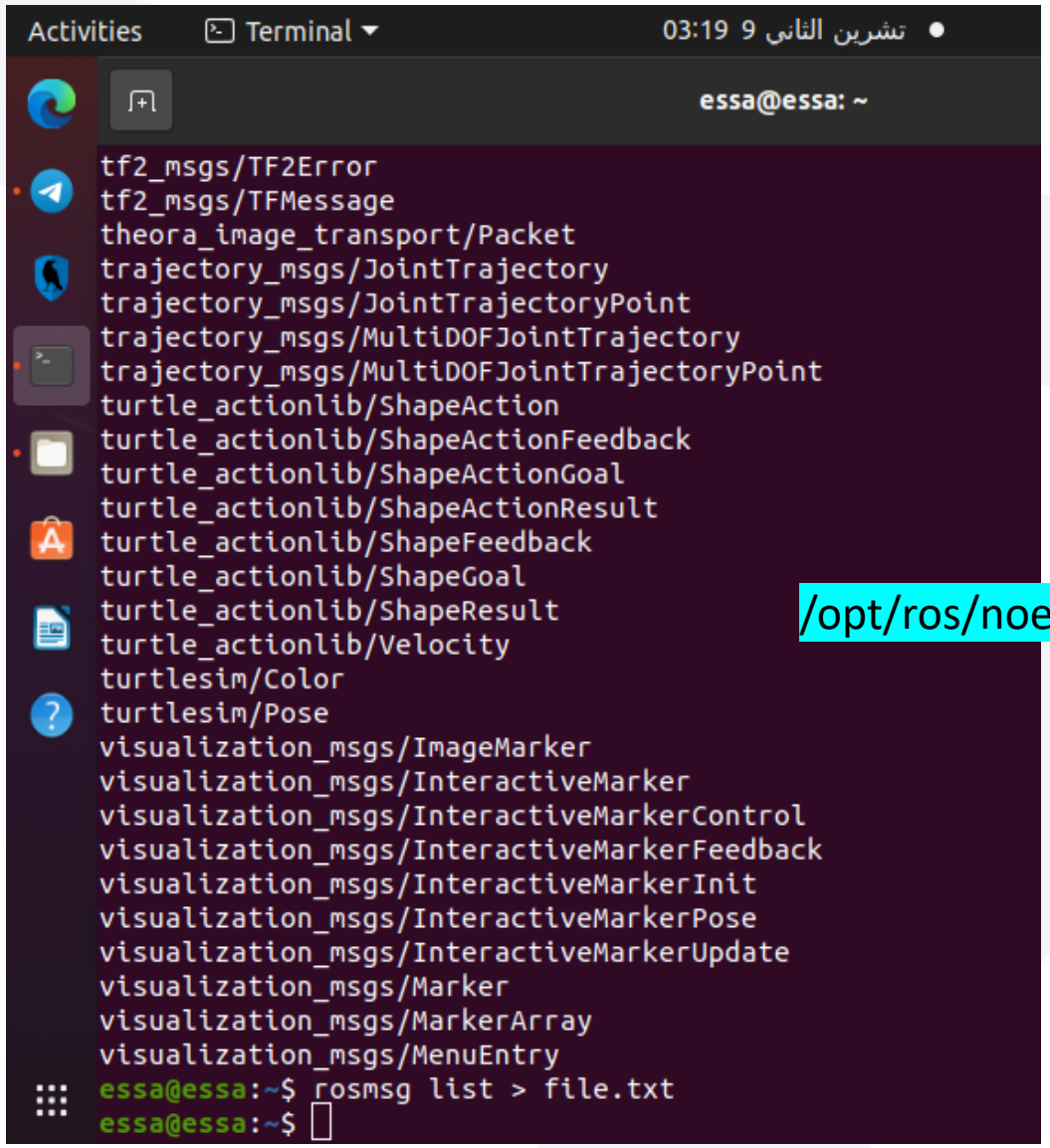
rosmsg packages Int32

actionlib
actionlib_msgs
actionlib_tutorials
bond
control_msgs
controller_manager_msgs
diagnostic_msgs
dynamic_reconfigure
gazebo_msgs
geometry_msgs
map_msgs
nav_msgs
pcl_msgs
roscpp
rosgraph_msgs
rospy_tutorials

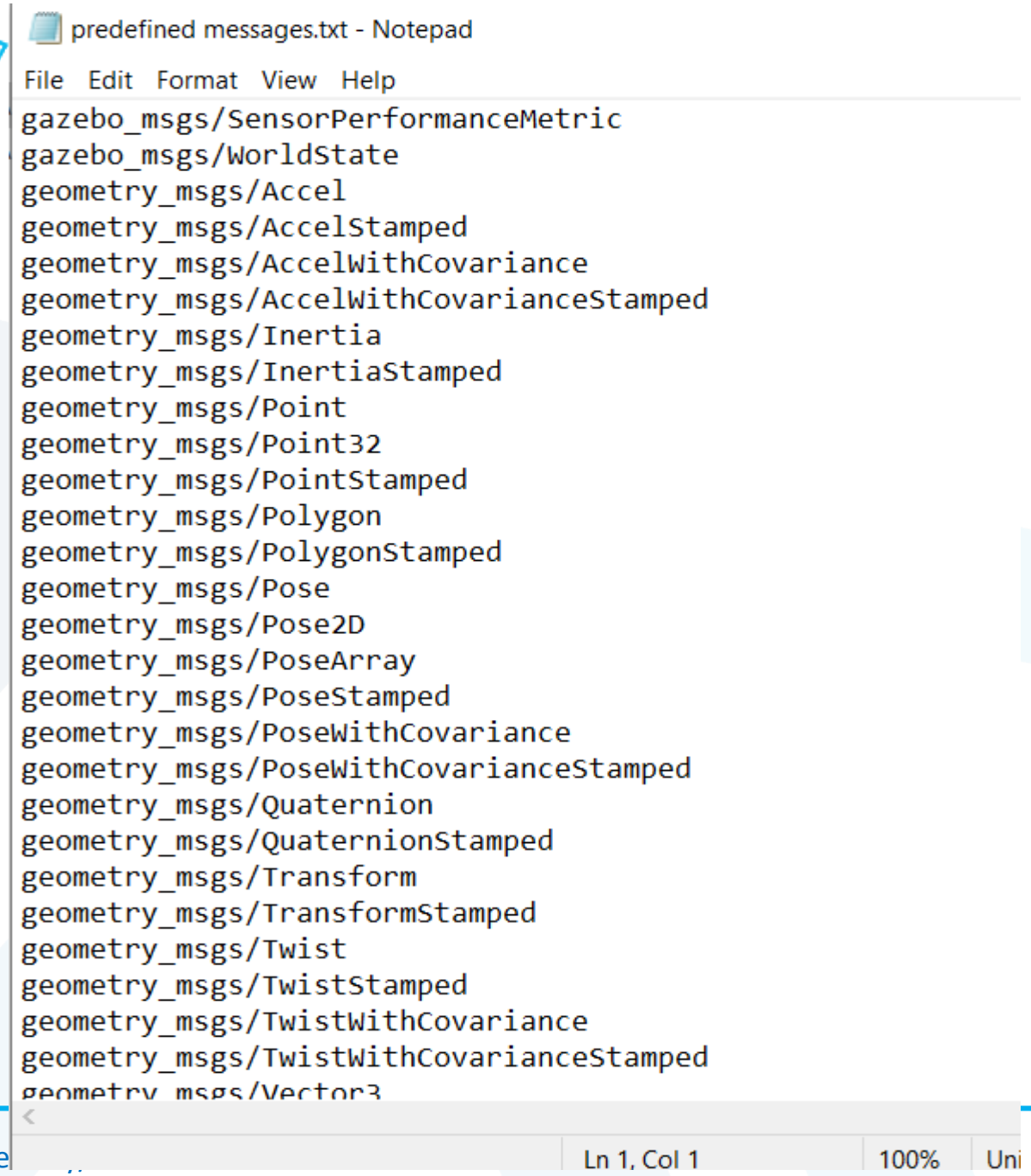
sensor_msgs
shape_msgs
smach_msgs
std_msgs
stereo_msgs
tf
tf2_msgs
theora_image_trar
trajectory_msgs
turtle_actionlib
turtlesim
visualization_msgs

rosmsg list: lists all the messages.

- rosmsg list > file.txt



/opt/ros/noetic/include



ROS Messages

- For the publisher (turtle_teleop_key) and subscriber (turtlesim_node) to communicate, the publisher and subscriber must send and receive the same type of message.
- This means that a topic type is defined by the message type published on it.
- The type of the message sent on a topic can be determined using: rostopic type.

rostopic type [topic]

$ rostopic type /turtle1/cmd_vel

geometry_msgs/Twist

This prints the topic type (the type of message it publishes).

$ rosmsg show geometry_msgs/Twist

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

to see the message fields

rostopic continued

Using rostopic pub

rostopic pub publishes data (messages) on to a topic currently advertised.

Usage:

```
rostopic pub [topic] [msg_type] [args]
```

```
$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
```

The previous command will send a single message to turtlesim telling it to move with a linear velocity of 2.0, and an angular velocity of 1.8 .

- rostopic pub: This command will publish messages to a given topic:
- -1 : This option (dash-one) causes rostopic to only publish one message then exit:
- /turtle1/cmd_vel: This is the name of the topic to publish to:
- geometry_msgs/Twist: This is the message type to use when publishing to the topic:
- -- This option (double-dash) tells the option parser that none of the following arguments is an option. This is required in cases where your arguments have a leading dash -, like negative numbers.
- As noted before, a geometry_msgs/Twist msg has two vectors of three floating point elements each: linear and angular. In this case, '[2.0, 0.0, 0.0]' becomes the linear value with x=2.0, y=0.0, and z=0.0, and '[0.0, 0.0, 1.8]' is the angular value with x=0.0, y=0.0, and z=1.8. These arguments are actually in YAML syntax

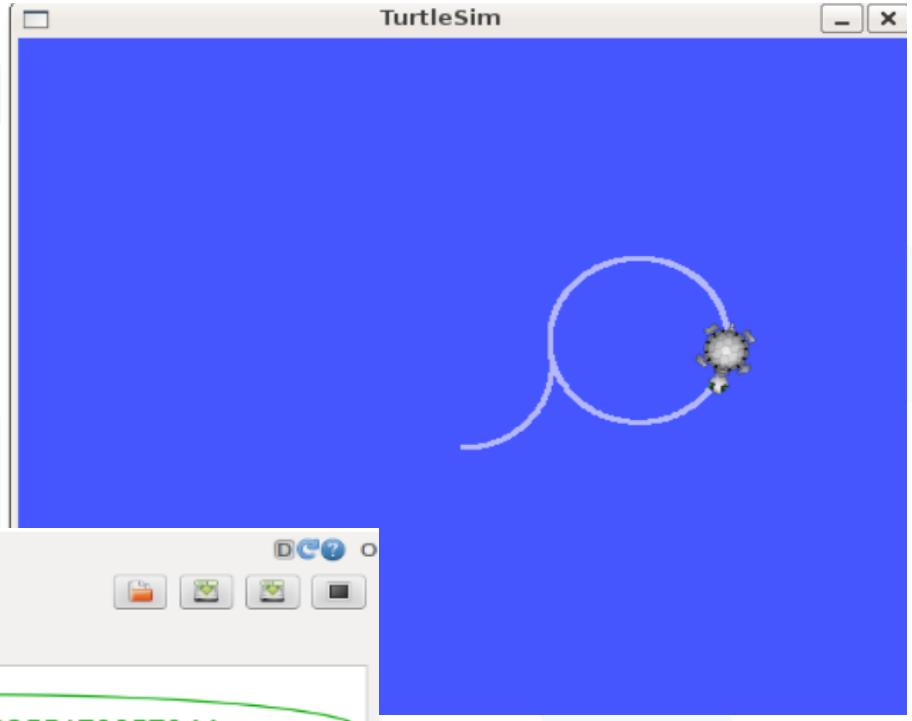
rostopic pub [topic] [msg_type] [args]

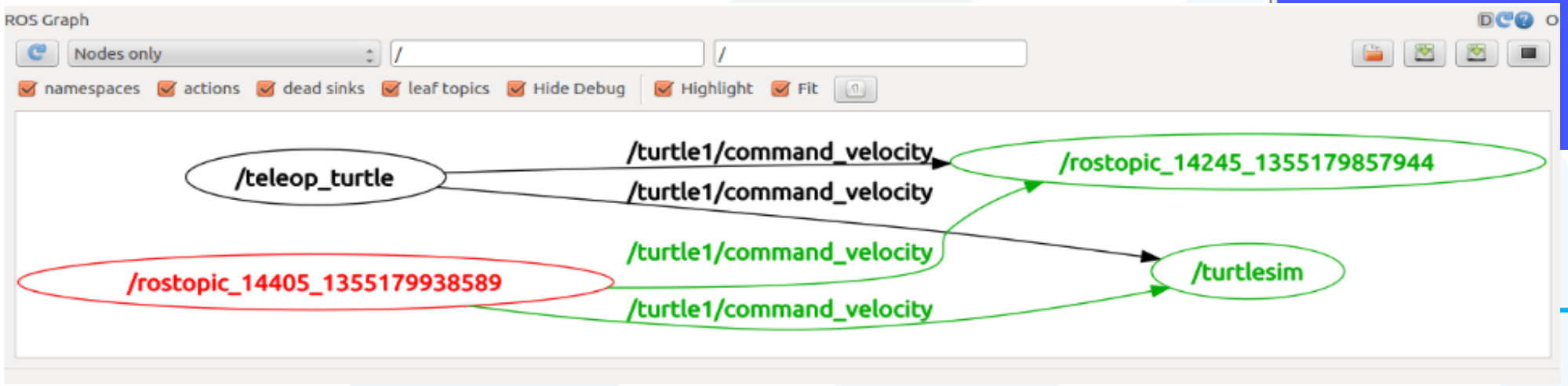
rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'

To keep moving, the turtle requires a steady stream of commands at 1 Hz.

We can publish a steady stream of commands using rostopic pub -r

This publishes the velocity commands at a rate of 1 Hz on the velocity topic.





```
essa@essa:~$ rostopic type /turtle1/pose
turtlesim/Pose
essa@essa:~$ rosmsg show turtlesim/Pose
float32 x
float32 y
float32 theta
float32 linear_velocity
float32 angular_velocity
```

```
essa@essa:~$ rostopic echo /turtle1/pose
x: 0.0004145863349549472 3
y: 7.335521221160889
theta: -1.5500030517578125
linear_velocity: 0.0
angular_velocity: 0.0
---
x: 0.01226711273193359 4
y: 7.344272136688232
theta: -1.422003149986267
linear_velocity: 0.0
angula
```

ROS Services

- Services: when we publish topics, we are sending data in a many-to-many fashion, but when we need a request or an answer from a node, we can't do it with topics.
- Services are another way through which nodes can communicate with each other.
- Services allow nodes to send a request and receive a response
- Service (srv) types: define the request and response data structures for services in ROS.
- The services are developed by the user, and standard services don't exist for nodes.

rosservice tools

Using rosservice

rosservice has many commands that can be used on services, as shown below:

Usage:

- rosservice list: This lists the active services.
- rosservice call /service args: This calls the service with the provided arguments.
- rosservice type /service: This prints the service type.
- rosservice find or rosservice find msg-type: This finds services by the service type.
- rosservice info /service: This prints information about the service.
- rosservice uri /service: This prints the service ROSRPC URI.

rosservice list

$ rosservice list

The list command shows us that the turtlesim node provides **nine** services: reset, clear, spawn, kill, turtle1/set_pen, /turtle1/teleport_absolute, /turtle1/teleport_relative, turtlesim/get_loggers, and turtlesim/set_logger_level.

There are also two services related to the separate rosout node: /rosout/get_loggers and /rosout/set_logger_level.

list the services available for the turtlesim node:

1. /clear
2. /kill
3. /reset
4. /rosout/get_loggers
5. /rosout/set_logger_level
6. /spawn
7. /teleop_turtle/get_loggers
8. /teleop_turtle/set_logger_level
9. /turtle1/set_pen
10. /turtle1/teleport_absolute
11. /turtle1/teleport_relative
12. /turtlesim/get_loggers
13. /turtlesim/set_logger_level

rosservice type

If we want to see the type of any service, for example, the /clear service, we use: rosservice type

Usage:

rosservice type [service]

$ rosservice type /clear

std_srvs/Empty

This service is empty, this means when the service call is made it takes no arguments (i.e. it sends no data when making a request and receives no data when receiving a response).

rosservice call

To invoke a service, we will use: rosservice call

Usage:

rosservice call [service] [args]

To invoke the /clear service ,we use:

$ rosservice call /clear

Here we'll call with no arguments because the service is of type empty

This does what we expect, it clears the background of the turtlesim_node (remove all path lines).

to try another service, for example, the /spawn service

This service will create another turtle in another location with a different orientation

$ rosservice call /spawn

```
Usage: rosservice call /service [args...]

rosservice: error: Please specify service arguments
```

Rosservice tools

Let's look at the case where the service has arguments by looking at the information for the service spawn:

```
$ rosservice type /spawn | rossrv show
float32 x
float32 y
float32 theta
string name
---
string name
```

With these fields, we know how to invoke the service

We need the positions of x and y, the orientation (theta), and the name of the new turtle. The name field is optional

```
$ rosservice call /spawn 2 2 0.2 ""

name: turtle2
```

This service lets us spawn a new turtle at a given location and orientation.
The service call returns with the name of the newly created turtle

so let's not give our new turtle a name and let turtlesim create one for us.

```
$ rosservice call /spawn 3 3 0.2 "new_turtle"
```

Example:

- essa@essa:~/mycatkin_ws$ cd src/
- catkin_create_pkg myturtlepackage rospy std_msgs geometry_msgs
- cd ..
- catkin_make

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
def move_turtle(linear_velocity, angular_velocity):
pub = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size=10)
    rospy.init_node('mynode') # Node name is 'mynode'
    rate = rospy.Rate(10) # 10Hz
        while not rospy.is_shutdown():
        move_cmd = Twist()
        move_cmd.linear.x = linear_velocity
        move_cmd.angular.z = angular_velocity
        pub.publish(move_cmd)
        rate.sleep()
```

```
"""
    Moves the turtle in turtlesim based on the provided linear and angular velocities.
    Args:
        linear_velocity (float): The linear velocity in m/s.
        angular_velocity (float): The angular velocity in rad/s.
    """
```

```
if __name__ == '__main__':
    try:
        linear_velocity = 0.5 # m/s
        angular_velocity = 0.2 # rad/s
        move_turtle(linear_velocity, angular_velocity)
    except rospy.ROSInterruptException:
        pass
```

جَامعة المَنارة
MANARA UNIVERSITY

شكرا لحسن الاصغاء