Code and Architecture Example

1 Relating your first run to the publisher subscriber model

Remember to run in each terminal you open:
1. cd catkin_ws
2. source devel/setup.bash

Roscore- Running roscore in a separate terminal will set up your ros master node for the system. The ros master node, is a node that manages all other nodes and topics created by the user.

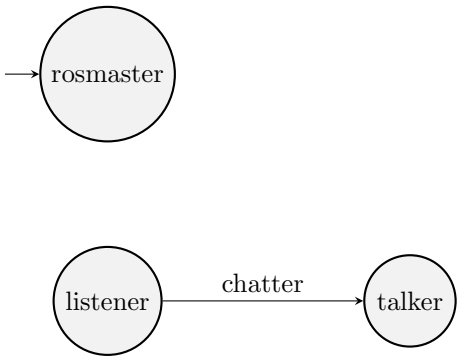
rosrun- This command runs and adds a node to your architecture.

examples(run each in a new terminal):
rosrun talk_listen listener.py
rosrun talk_listen talker.py

The argument test_run is a package, for now think of a package as a set of nodes that defines an architecture.

listener.py is the python script that runs the node listener, which subscribes to topic chatter.

talker.py is the python script that runs the node talker, which publishes to the topic chatter.
(code provided on blackboard please review!)



To view this architecture in ros(for debugging and making sure everything is correct) run the command:
rosrun rqt_graph rqt_graph

2 Turtlesim

First lets start by running roslaunch command. The roslaunch command allows you to launch multiple nodes and the rosmaster with just one command.
example: *roslaunch turtle_motion turtle_clean.launch*
(remember to create the package as instructed in the ROS_Instructions pdf available on blackboard.)

If nothing happens on the turtle simulator, open up the file catkin_ws/src/turtle_motion/src/grid_clean.py and call any of the functions included from there in the main function.

This command will run your rosmaster, rosrun turtlesim turtlesim_node, and rosrun turtle_motion grid_clean.py all in one terminal and you do not have to worry about managing multiple nodes across multiple terminals.

turtle_clean.launch ×

catkin_ws > src > turtle_motion > launch > turtle_clean.launch

```
1  <launch>
2      <node pkg="turtlesim" type="turtlesim_node" name="turtlesim_node" output="screen"/>
3      <node pkg="turtle_motion" type="grid_clean.py" name="grid_cleaner_node" output="screen"/>
4  </launch>
```

The important things to note about this file is that you must start and end the file with the launch tag (<launch>). Each node you set up begins with a node tag(<node>) and then you must specify the package where the node is contained(pkg=), the name of the file that sets up the node(type=), and the name of the node(name=); all other arguments are optional.

```
################################################################################
if __name__ == '__main__':
    try:
        rospy.init_node("grid_clean", anonymous=True)
        vel_pub = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size=10)
        pose_sub = rospy.Subscriber('/turtle1/pose', Pose, poseCallback)
        time.sleep(2)

    except rospy.ROSInterruptException:
        rospy.loginfo("node terminated")
```

The image above shows how we set up the node, publisher and subscriber. Note that every time you set up a publisher or subscriber you must both provide a topic name and a message type.Example: rospy.Subscriber(topic_name, msg_type, callback function name). The callback function is written by the programmer, and it is the code that is executed every time the node receives a message on a topic.

```
def poseCallback(pose_message):
    global x, y, yaw
    x = pose_message.x
    y = pose_message.y
    yaw = pose_message.theta
```

Example of callback in code that updates variables x, y, and yaw based on the robots current position on the simulator.

```
def move(speed, distance, is_forward):
    vel_msg = Twist()
    x0 = x
    y0 = y

    if is_forward:
        vel_msg.linear.x = abs(speed)
    else:
        vel_msg.linear.x = -abs(speed)

    distance_moved = 0.0
    loop_rate = rospy.Rate(10)

    while distance_moved < distance:
        vel_pub.publish(vel_msg)
        loop_rate.sleep()
        distance_moved = abs(math.sqrt(((x-x0)**2) + ((y-y0)**2)))
```

The move function has the robot move at some speed linear.x by publishing to the /turtle1/cmd_vel topic. The message type of this topic is geometry_msgs/Twist. This message has two member variables Vector3 linear and Vector3 angular. The linear variable sets the linear velocity of the turtle and the angular variable sets the angular velocity of the turtle robot. In the code we use the distance formula to compute how much the robot has moved from its initial position.

```
def rotate(angular_speed_degree, relative_angle_degree, clockwise):
    vel_msg = Twist()
    angular_speed = math.radians(abs(angular_speed_degree))

    if clockwise:
        vel_msg.angular.z = -abs(angular_speed)
    else:
        vel_msg.angular.z = abs(angular_speed)

    loop_rate = rospy.Rate(10)
    current_angle_degree = 0.0
    t0 = rospy.Time.now().to_sec()

    while current_angle_degree < relative_angle_degree:
        vel_pub.publish(vel_msg)
        loop_rate.sleep()
        t1 = rospy.Time.now().to_sec()
        current_angle_degree = (t1-t0)*angular_speed_degree
```

The rotate function also publishes /turtle1/cmd_vel topic and uses a message type of geometry_msgs/Twist. This time we are interested in the member variable Vector3 angular which sets the rotational velocity of the robot. The speed input parameter tells us how fast our turtle robot will rotate per second,

and the idea is to rotate for however many seconds are required to reach our goal.

2.1 turtle architecture