Intro to ROS - 2nd exercise

May 2023

Course:
Exercises - Lecture 3 **Lecturer:**
Prof. Markus Ryll **Exercise**
Material:
Jon Arrizabalaga (jon.arrizabalaga@tum.de)

This document proposes three exercises to assist students in understanding the content presented in Lecture 3 of ”Introduction to ROS”. If completed, it is guaranteed that the students have acquired the necessary knowledge for following the upcoming lectures.

Notice1: There are infinite correct solutions! Coding is like writing, you can express the same idea in an infinite amount of ways!

Notice2: For exercises 1 and 2, modifying the *listener* and *talker* roscpp tutorials is enough.
Tutorial: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29

Notice3: In exercise 3, if you are clueless, do not panic. Checking the provided solution template will get you rolling!

1 Scenario

For this exercise we will use the same set-up as the one in the lecture (see Figure 1). There is a town called *Rosheim*, whose inhabitants either work in a supermarket, *SupeROS*, or are *Customers*. The food sold in the supermarket is divided into three sections, each of which is overseen by an employee: *MrFish* is the responsible for *fish*, same for *MrVeggies* with *veggies* and *MrFruit* with *fruit*. From the *Customers* side, there are two people: *MrCustomer1* and *MrCustomer2*, each being interested in a given *product*.

If we convert this to ROS-terms:

> ”For this exercise we will use the same set-up as the one in the lecture (see Figure 1). There is a **workspace** called *rosheim ws*, with two **nodes** *SupeROS* and *Customers*. The data published in the **node** *SupeROS* is divided into three **topics**, each of which is published by a **publisher**: *MrFish* is the publisher for topic */fish*, same for *MrVeggies* with */veggies* and *MrFruit* with */fruit*. In the **node** *Customers*, there are two **subscribers**: *MrCustomer1* and *MrCustomer2*, each of them subscribing to a given **topic** according to their product interests.”

2 Exercise 1

Create a package (supeROS pkg), containing a node, *supeROS*, with three **publishers** publishing to three different topics, where

> *mrFish* is the publisher for topic */fish* *mrVeggies* is
>
> the publisher for topic */veggies*

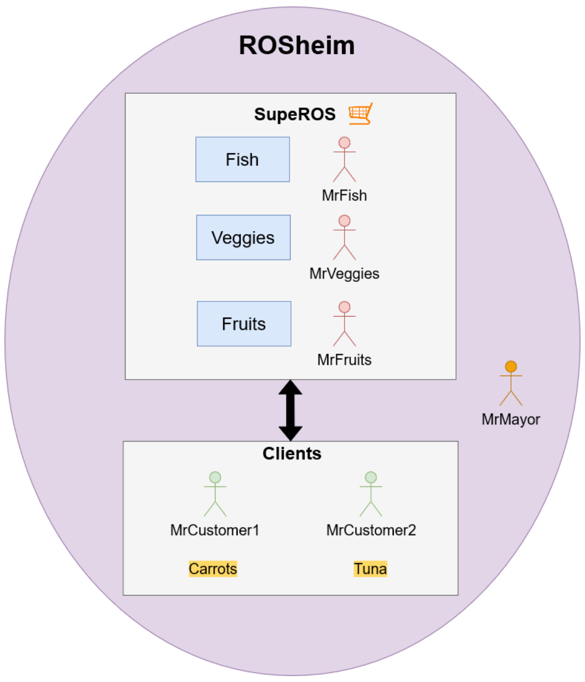

Figure 1: Environment used for exercises

mrFruit is the publisher for topic */fruit*

. For all of them, the message type is std msgs::String and contains the name of the product. The products are offered according to a list that restarts when all of them are published. The available products for every section are:

/fish: tuna, salmon, shark

/veggies: onion, potatoes, carrots

/fruit: bananas, apples, grapes

The message type for every publisher is std_msgs::String and contains the name of the product. The products are offered according to the lists above. Notice that the broadcasting is infinite, meaning that it is restarted when all of them are published, i.e., the output of topic /fish should be [tuna, salmon, shark, tuna, salmon, shark, tuna, ...]

If compilation works and you finished writing the node, run rostopic echo /veggies (or any other topic), and you should see the available products printed in the command window at a very high frequency.

3 Exercise 2

Having created the supermarket, lets tackle the clients. Create another package (customers pkg), containing node *customers* with two subscribers, subscribed to the food category respective to the product they wish:

1. Customer1: carrots (topic /vegetables)
2. Customer2: tuna (topic /fish)

Inside the callback function of each subscriber it is verified if the desired product is equal the one being shouted by the publishers of Exercise 1. If this condition is fulfilled, you have to print ”I purchased Y!”, where Y is the product’s name.

To verify if you succeeded, run it alongside the node in Exercise 1. You should see the prints of the customers in the command window.

4 Exercise 3

Purpose: At this point you might have noticed that subscriber callback functions only have a single input, which is the data received from the topic to which they are subscribed. Since we might want to use variables that have been defined outside the callback function, it is highly recommended to write nodes with an object oriented programming (OOP) approach, by embedding all the variables in a class. For example, in the previous exercise, we wanted to double check if the message received from the network was equivalent to the interest of the customer, i.e, if (carrots==interest customer){print ...}. To overcome this problem, we have declared interest customer inside the callback function. As you will see, this is not always possible, leaving no other option than writing the node in an OOP style.

In this exercise, you will create another node ”customers _money” inside the already existing package ”Customers pkg”. The scenario and purchase process is the same as in Exercise 2. However, in this case, once the customer receives the product, he pays back with a coin. To make it simple, all products will be worth 1 coin. To do so, you will have to create a publisher that publishes to topic /Money. The message will be of type std msgs::Float. As soon as the consumer verifies that the announced product is equivalent to the desired one, he/she will have to pay back and print ”I purchased Y and I already paid back!”, where Y is the product’s name. The scheme below might help

```
void customerCallback(… msg)
{ if (msg.data == desired_product_) //check if available product is the desired one {

    //pay back (coin is type float and should contain the prize of the product.)
    customer_publisher_.publish(coin)

    //print in command window
    ROS_INFO("I purchased %s and I paid back!", desired_product_.c_str()); }
}
```

From the code above, you might have already realized that inside the callback function we are using customer_publisher_. To do so, this variable requires to be previously declared. Doing it inside the function would cause it to be initialized every time the callback is invoked, which is a very inefficient approach that slows down the callback’s completion. Therefore, the publisher should only be initialized once the node is started. Since the initialization takes place outside the scope of the callback function, we have to do something to guarantee that we will be able to access it. The standard way to solve this is by creating a class where all required variables (publishers, subsribers,...) are member-variables. In other words, since we have acces to all member-variables, the aforementioned limit of a unique input of the subscriber callback becomes irrelevant. This means that we will be able to call the publisher from the subscriber’s callback.

The verification process is the same as before. On top of that, you can check that the existence of the new topic /Money by running rostopic list. You can also see the coins being sent by running rostopic echo /Money

If you do not know how to start, take a look at the template!