Institute for Cognitive Systems
Technische Universität München
Prof. Gordon Cheng

Multi-sensory Based Robot Dynamic Manipulation
ROS Tutorial 2: Getting familiar with ROS

1 ROS basic toolbox

a Creating and setting up a ROS workspace

a.1 How to create a new ROS workspace and initialize it using catkin

Start by creating a general folder for all the workspaces you will be developing:

```
$ cd                        #Move to the home directory
$ mkdir -p ros/workspaces #Create a new directory for your ROS workspaces
$ cd ros/workspaces       #Move to this subdirectory
```

Within this folder, create a new ROS workspace and initialize it:

```
$ mkdir -p my_workspace/src #Create your workspace directory
$ cd my_workspace/src #Switch to the src subfolder.
$ catkin_init_workspace      #Initialize your new workspace
```

The `catkin_init_workspace` command will create the `CMakeLists.txt` file (actually it just create a symbolic link in the “`src`” folder of your workspace pointing to:
`CMakeLists.txt` $\rightarrow$ `/opt/ros/indigo/share/catkin/cmake/toplevel.cmake`

a.2 How to build your ROS workspace

Once your workspace is created, you have to compile it:

```
$ cd ~/ros/workspaces/my_workspace
$ catkin_make #Compile your workspace
```

Even though the workspace is empty (there are no packages in the “`src`” folder, just a single `CMakeLists.txt` symbolic link) you can still build it. If you now look at your root directory, you should now have a “`build`” and a “`devel`” folder. The “`build`” folder mainly contains executables of the nodes that are placed inside the catkin workspace “`src`” folder. The “`devel`” folder contains bash script, header files, and executables in different folders generated during the build process. After building the empty workspace, we should set the environment of the current workspace to be visible by the ROS system. This process is called **overlaying a workspace**. Inside the “`devel`” folder you can see that there are now several “`setup.*sh`” files. Sourcing any of these files will overlay this workspace on top of your environment so that you can use it:

```
$ cd ~/ros/workspace/my_workspace
$ source devel/setup.bash #Source your workspace so that ROS is aware of it
```

To make sure your workspace is properly overlayed by the setup script, make sure `ROS_PACKAGE_PATH` environment variable includes the directory you're in:

```
$ echo $ROS_PACKAGE_PATH
#Should give something like:
#/home/youruser/ros/workspaces/my_workspace/src:/opt/ros/kinetic/share
```

a.3 How to get QtCreator to work with your ROS workspace

QtCreator is a very powerful IDE, with advanced debugging features. It is free and can be installed from the official ubuntu repository using the following command line:

```
$ sudo apt-get install qtcreator
```

QtCreator can be easily interfaced with a ROS workspace provided that the associated *CMakeLists.txt* file is no longer a symbolic link:

```
$ cd ~/ros/worspace/my_workspace/src
$ mv CMakeLists.txt CMakeLists.txt.old
$ cp CMakeLists.txt.old CMakeLists.txt
```

Open Qtcreator and make it point to your "`src`" folder:

```
$ cd ~/ros/worspace/my_workspace/src
$ qtcreator CMakeLists.txt & #Start QtCreator as a background task
```

b Creating a ROS package

Packages are the most basic unit of the ROS middleware. They contain the ROS runtime process (called nodes), libraries, configuration files, headers and so on, which are organized together as a single unit. The `catkin_create_pkg` command is used to create a ROS package. It has the following syntax:
`catkin_create_pkg [package_name] [dependency 1] [dependency 2] ... [dependency n]`
You can generate a new ROS package within the previously created workspace:

```
$ cd my_workspace/src #Switch to the src subfolder.
$ catkin_create_pkg my_package std_msgs rospy roscpp #Create a new package
```

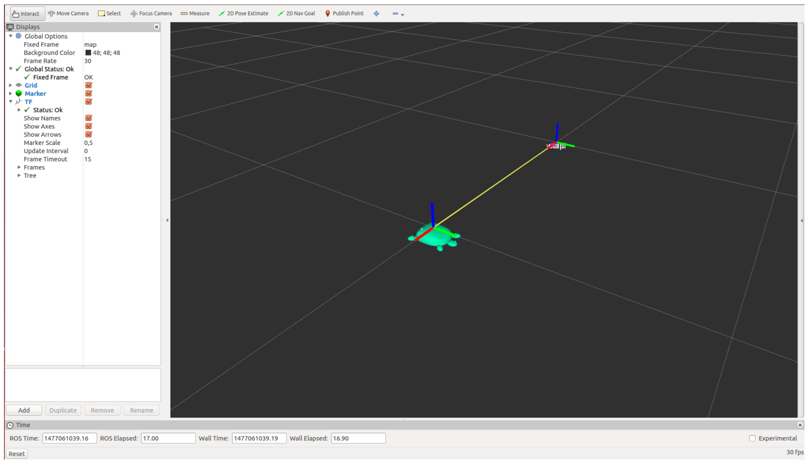
In this example, the dependency are as follows:

- **roscpp**: is a ROS library which provides APIs to C++ developers to make ROS nodes with ROS topics, services, parameters, and so on. You must include this dependency if you want to write a ROS C++ node.
- **rospy**: is the equivalent of roscpp for Python.
- **std_msgs**: contains basic ROS primitive data types such as integer, float, string, array, and so on. We can directly use these data types in our nodes without defining a new ROS message.

After creating this package, build the package without adding any nodes using the `catkin_make` command. After a successful build, we can start adding nodes to the "`src`" folder of this package.

2 Homework

The objective of this tutorial is to pilot a “turtlebot” robot in the *rviz* 3D visualization environment, using the information flux generated by your keyboard. At the end of this tutorial, you should be familiar with some of the most important concepts of ROS, namely packages, publishers, subscribers and of course *rviz*. Use the provided template to complete the exercises.



Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
Marker	✓
TF	✓
✓ Status: Ok	
Show Names	✓
Show Axes	✓
Show Arrows	✓
Marker Scale	0,5
Update Interval	0
Frame Timeout	15
Frames	
Tree	

ROS Time: 1477061039.16 ROS Elapsed: 17.00 Wall Time: 1477061039.19 Wall Elapsed: 16.90

Figure 1: Objective of this tutorial

You should deliver a compressed `.zip` file with your implementations using the instructions of the exercise sheet and of the given template. Include a `Readme.txt` file in order to indicate how to run your nodes and to answer the questions. Please, name the compressed file as follows: "`Name_lastName_MSRDM_ROStutorial2.zip`'. Example of the `Readme.txt` file:

```
1) roslaunch turtle_vis TurtleVis.launch
2) rosservice call /TurtlePose "p:
x: 2.0
y: 2.0
theta: 1.57"
3) rosrun turtle_vis turtle_set_position_node
(cm, cm, rad)
```

This tutorial will have to be delivered no later than the **Wednesday 15/11/2017, 23h59**.
Important: This tutorial is strictly personal. You can of course discuss with your classmates about obtained results but the submitted code should be different for every student. The code similarity will be checked at the end of the semester, when the tutorial will be graded.

a Before you begin

1. Create a new ROS workspace and name it using the following convention:
   `MSBRDM_tutorial_2_YOURNAME`.

2. Extract the template package folder into the “src” folder of your ROS workspace and build it:

```
$ cd ~/ros/worspace/MSRDM_tutorial_2_YOURNAME
$ catkin_make
```

It does not compile ? This is perfectly normal since the makefile inside the template is incomplete: your first exercise is to fix it...

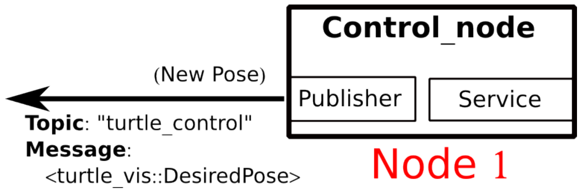
b Exercice 1: Fix the makefile of your package

Fix the CMakeList.txt of the template project (Hint: Look for the **#TODO** inside the file and take a look at: `http://wiki.ros.org/catkin/CMakeLists.txt`)

1. Include new system packages
2. Add the new defined messages
3. Include the required service files
4. Include the name of the new defined library on `catkin_package`
5. Add the nodes that will be executed and create the proper target link.

c Exercice 2: Create a kinematic control node

The goal of this exercise is to create a node which will compute the new turtle pose using a simple Kinematic Control.



The Node 1 is named `Turtle_Control_node`. It must provide:

- a **service**, is used to receive the desired turtle poses (x_d, y_d, θ_d).
- a **publisher**, which will publish a topic "`turtle_control`" containing the new turtle pose. This pose will be listened by the Node 3.

Use the provided template "`turtle_control_node.cpp`" to create node 1 and look for all **#TODO** flags. Your task inside the main loop of "`turtle_control_node.cpp`" is basically to:

1. Obtain the desired pose $\mathbf{x}_d$ from a class variable.
2. Implement a P-control (Kinematic control) to move the turtle to this desired position:

$$\boxed{\mathbf{v} = -K_p \cdot (\mathbf{x}(t) - \mathbf{x}_d)} \quad (1)$$

where the current position $\mathbf{x}(t)$ of the turtle is given by: $\mathbf{x}(t) = \mathbf{x}(t-1) + \mathbf{v} \cdot \Delta t$

3. Publish the obtained turtle position to Node 3.

Hints:

- To set the gain values for the controller (K_p), create a `*.yaml` file, which is usually in a new folder inside your package e.g. `turtle_vis/configs`.
- Modify the launch file to set the ros parameters from the `*.yaml` file (see the template in `turtle_vis/launch`).
- Define a message type to send the new position of the turtle through a publisher topic to Node 3.
- Take a look at: `http://wiki.ros.org/rosparam`

d Exercice 3: Some general questions

Answer the following questions:

1. How can I send the new desired pose of the turtle (x_d, y_d, θ_d) to Node 1. Please, indicate the command that you will use and explain the reasons.
2. What is the main difference between a Publisher/Subscriber and Service/Client? For example, what would happen if I replace the Service for a Subscriber in Node 1?
3. Node 1 also has a service that will receive the desired position and orientation of the turtle.
4. Define a new message type for the service of Node 1.
5. Replace the “callback function” from the template file (TurtleClass.h and TurtleClass.cpp) with the name of your function for the service, this function should be defined in your class.

Hint: You can use the following command to debug the correct behavior of node 1, i.e. send the desired position of the turtle from the ros service terminal command, e.g.

```
$rosservice call /TurtlePose "p: x: 0.0 y: 0.0 theta: 0.0"
```

Take a look at: `http://wiki.ros.org/rosservice`

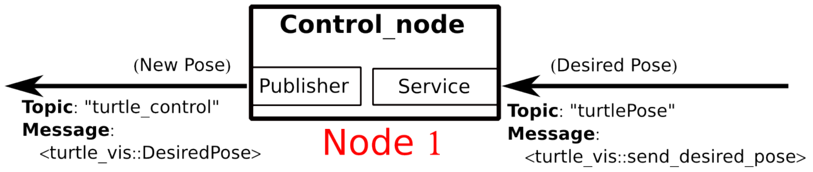


Figure 2: Exercice 3

e Exercice 4: Create a desired trajectory node

Create the node 2 in order to set the new desired pose of the turtle (x_d, y_d, θ_d) from a client to your defined service from node 1 . Node 2 is named: `Turtle_set_position_node` and it contains:

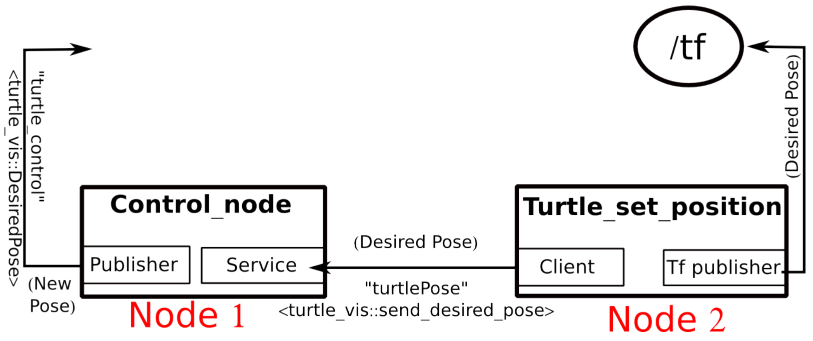


Figure 3: Exercice 4

- a **client** which will connect to the service provided by Node 1 and will send the desired turtle pose (acquired from the terminal (x_d, y_d, θ_d) in a continuous loop)
- a **TF Publisher** which will publish the coordinate frame of the desired turtle pose to the `/tf` node.

You must define a custom message. This message will be used for the topic for the Service/Client definition (communication between Node 1 and Node 2), e.g. `turtle_vis::send_desired_pose`.

f Exercice 5: Visualization node

Create a Node 3 that receives the new computed pose of the turtle from Node 1 and visualize the turtle with its new pose using rviz. Node 3 is named: `Turtle_vis_node`. I must provide:

- a **subscriber**, which will connect to the topic generated by Node 1 and will generate the coordinate frame for the current turtle position and the visualization of the turtle.
- a **TF Publisher** to publish the coordinate frame for the current turtle pose.
- a **Visualization_Marker Publisher** to visualize the turtle mesh in rviz.

As additional nodes, we have static transformations as well as rviz transformations using tf:

- Node 4 (Static transformation) perform a static transformation between the coordinate frames `/map` and `/world`. This `tf` will publish the *rviz* visualization.
- Node 5 (rviz visualization) allows to visualize the turtle mesh (marker), the world coordinate frame (tf), the turtle tf and the desired pose tf.

g Exercice 6: Make your system modular

Make your system modular, i.e. use a common object class for the subscriber (Node 3) and the client (Node 2) callback functions. Look at the file: `turtle_vis/src/solutions/myClass`

1. Create a callback function for the service in Node 1
2. Create two methods for obtaining the turtle pose needed in Node 1

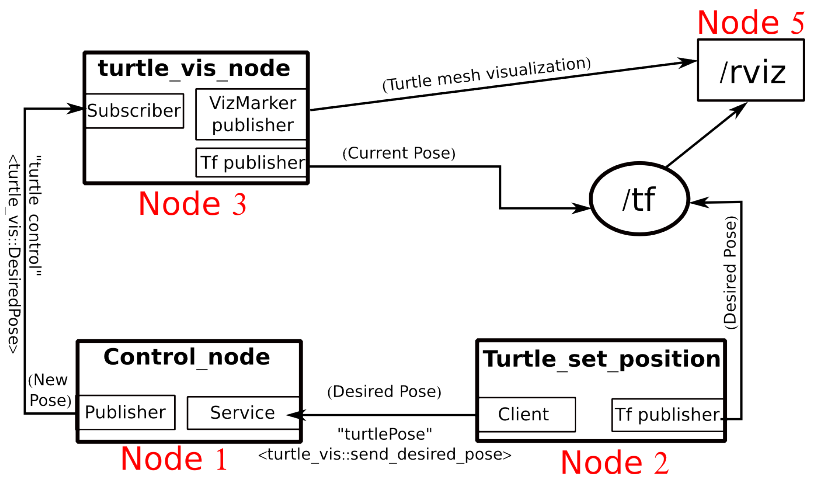
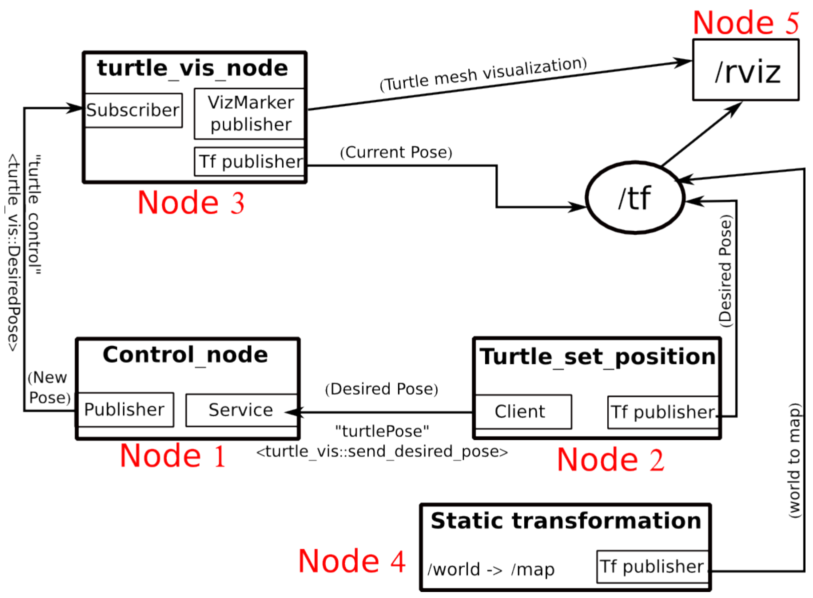


Figure 4: You must define a custom message. This message will be used for the topic publisher/-subscriber (communication between Node 1 and Node 3), e.g: `turtle_vis::DesiredPose`



3. Create a callback function for the subscriber of Node 3

Important: Create the header for the class in a separate file and place it in `turtle_vis/include/turtle_vis/myClass`