ROS – Lab 3

Goal
Learn to use services.
Learn to use the tf package.

Content
In this lab, we will use a very simple simulation of a planar dual arm robot. Each arm of the robot is a planar RR robot. The goal is to maintain the end point of the right arm at the same (x,y) position as the left arm end point, as far as the reach and joint limits of the right arm allows.
To achieve this goal we will do the following:

- Attach a frame to the left forearm, at the position we want to reach with the right arm. To do this we will use a tf_broadcaster, which is a node which adds a frame to the tree of transformations.
- Use the tf package to calculate the position of that frame with respect to the base of the right arm.
- Use the Inverse Kinematics service of the right arm to calculate the solution(s) to the IGM if any.
- Send the robot a joint position that reaches the desired spot if there is one.

You have a short video provided with the packages which shows the final result of the lab.

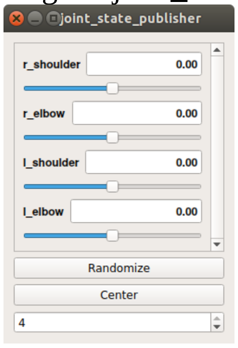
Remarks

- For the whole lab, the organization in packages is provided. So are the node skeletons, which are already named: you just need to fill the code. This is done so you don't need to worry about anything concerning the CMakeLists.txt and package.xml files.
- This lab is a bit more demanding in terms of programming. Those of you with less command of C++ will have more difficulties. I tried to give you important pointers, though. But to be honest, if you have problems with this level of C++ programming, then you have a problem that you must solve…
- Submit to Hippocampus at the end of each step (except step 1 of course).

Task 1: Setting up the environment.

The software is given as a meta-package called dual_planar_arm.

- Extract the meta-package to your ros/src folder (or other location depending on your installation).
- Compile the code. Remember to use either catkin_make or catkin build, as you do usually and to never mix the two commands.
- Launch the simulator.launch file from package dual_planar_arm. Rviz will allow you to check all the frame names, which are displayed by default. You will be able to check the joint names and move the joints using the joint_state_publisher (figure below).



r_shoulder	0.00
r_elbow	0.00
l_shoulder	0.00
l_elbow	0.00

4

The joint_state_publisher window.

Task 2: Creating the tf_broadcaster node.

To create a tf_broadcaster, your internet search will typically take you to the following ROS tutorial page: http://wiki.ros.org/tf/Tutorials/Writing%20a%20tf%20broadcaster%20%28C%2B%2B%29
This tutorial is dubious on several aspects, but it contains the technical elements you need. That said:

- The example given shows the broadcasting done in the callback to a certain topic. Although it is technically possible, there is absolutely no reason to do so. By doing so, your broadcaster depends on some other node running, plus you do not get to choose the frequency of execution of your broadcaster. **Do not do that**. Just define a node that subscribes to nothing and broadcasts. You node will not **explicitly** publish to anything but will indeed publish to a topic called "/tf", as you will be able to see in the rqt_graph.
- The broadcaster object must not be defined inside the while loop of the node, but in the initialization part, like publishers, subscribers, etc. Indeed, creating a tf_broadcaster implies quite a bit of overhead, so you would have the same problems I described in the class if you try to create a new one each time. That's why in the example of the tutorial the broadcaster object is declared as **static**. If you do as I say, you have no reason to declare it static.
- Remember that, even though you do not use any node handle in your code, you still have to declare at least one.

In the sendTransform method, the last two arguments are strings. The last one is the name of the new frame. You can call it "goal" for exampe, as it will be the goal of the right arm. The previous is the name of the parent frame in the tf tree.

You can see the frame names in Rviz. You can also run the command "rosrun tf view_frames". A PDF file of the tree will be created in the folder where the command has been run. All the frame names are indicated.

The arm and forearm lengths is 1 m. The orientation of the frame has no importance, you can put roll, pitch and yaw angles to zero.

This task is very short. It should not take you more than 15 minutes.

Remarks

- There are simpler ways to broadcast a constant transform, that you may find on the internet. I want you to create your own tf_broadcaster in the way described above because it is the most general solution. In the future, if you need to broadcast a transformation which is time-varying, you will easily do it with this method, but not with the shortcut methods.
- In practice, in task 3, the same node could be both broadcaster and listener. But in general, it may not be the same node that broadcasts and listens. That's why I make you use a different node in task 3.

Task 3: Creating a tf_listener.

The basics are described here: http://wiki.ros.org/tf/Tutorials/Writing%20a%20tf%20listener%20%28C%2B%2B%29

You have to create a tf_listener which will provide you the position and orientation of the newly added frame "goal" with respect to the base frame of the right arm, called "r_arm_base". The position part of this information will later be used as input to the IGM of the right arm, thus giving us the joint configuration which allows both robot ends to coincide in the x-y plane.

A few tips:

- tf::StampedTransform is a class. There is no message type associated to it (if you type rosmsg show tf/StampedTransform it does not work. An object of this class has a position and orientation information. To access this information you need to use accessor methods, not something like object.position.x. See the link given above, or go to the tf::StampedTransform class reference page.
- Correctly format your code. I will **not** try to help you with programming errors if the code is not correctly formatted (indented, aligned begin and end of structures).
- Checking: Just output the x and y position and see if the values seem reasonable.

Task 4: Calling a ROS service to calculate the IGM of the RR robot (right arm).

Your node will be a ROS service client. I have written and provided the service server in the packages. You can review the class material about services. The main things to remember are:

- When your node calls a service, it waits for the result.
- Services use a specific message. Here the message type is defined in the dual_planar_arm_msgs package, in the srv folder. You can have a look, but you must not modify anything.
- Since services use a message type, there is a corresponding include instruction in the code.

You will need to have a look at `ros::ServiceClient` class reference and particularly methods `exists`, `call`. Do not use anything else. The logical order of the tests are:

- Check that the service exists.
- Call the service and check the return value. If the return value is false, it means that the request part of your message was incorrect. Here the request has a very simple format, so you should not have problems. There are cases where the messages are way more complex…

These checks should be in the while loop. You might think that it's enough to use the "exists" method in the initialization only. Not really, because the service is provided by another node, which may fail. If it does, all your calls will return "false" and you may believe that your request message is incorrect, when it's not. So it is better to perform the test inside the loop. That's why in my opinion method waitForService (see class reference) is pretty much useless.

The right arm IK service expects the request position to be, not in the base frame of the dual arm, but in the base frame of the right arm "r_arm_base".

Checking the result: print the joint configuration (1st solution in case there are two) if any or a message saying there is no solution. Check that the joint configuration looks reasonable. Printing the values in degrees helps (at least for me).

Task 5: Tracking the left arm end position with the right arm end position.

The last part of the work is to use the information provided by the IGM service to "control" the right arm. The result should be as in the provided video. Check the information about the simulator node to understand what topic you must publish to, and what type the topic is.

Use the launch file "complete.launch" to check your results.