Flight camp part 2

In this part, we will look at the ROS ecosystem and how to integrate larger robotic systems using ROS. ROS, or the "Robot Operating System," is a set of conventions and programs and is *not* an actual operating system like Windows or macOS. Typically, ROS is run on GNU/Linux and there is mild success with running ROS on Apple's macOS and there are efforts to containerize parts of it (e.g. using Docker). The choice is yours, but we assume you run on GNU/Linux, specifically Ubuntu 18.04 with ROS Melodic.

Basic ROS Concepts

Familiarize yourself with some basic ROS concepts, namely:

- **ROS Nodes (http://wiki.ros.org/Nodes)** are the bread and butter of ROS. Typically, a single node performs a single function, such as reading the camera, or sending actuator signals, or performing high-level planning. These are typically written in C++ using `roscpp`, or Python using `rospy`. In this tutorial we will use Python. Remember to **make your Python script executable (https://superuser.com/a/828740)**.
- **ROS Topics (http://wiki.ros.org/Topics)** are a hierarchical filesystem-like communications platform for ROS nodes. Nodes can *publish* data and *subscribe* to data -- for example, a camera node would publish camera images to some topic; a vision system would then subscribe to this camera topic. A single topic may have both many publishers, and many subscribers.
- **ROS Messages (http://wiki.ros.org/Messages)** are what nodes exchange on topics; each topic has a *single* message type. Message definitions are hierarchical.
- The **ROS Master (http://wiki.ros.org/Master)** or `roscore`, is the behind-the-scenes engine that coordinates topics, nodes, services, and so on. It is often implicitly started, but it is almost always a better idea to explicitly run a roscore separately.
- **ROS Packages (http://wiki.ros.org/Packages)** are software collections of nodes, message definitions, and build and install instructions. ROS packages also enable automatic discovery with a plethora of ROS-specific tools like `rosrun` and `roslaunch`.

These are the essential features of ROS, but we urge you to at least skim through the **Concepts (http://wiki.ros.org/ROS/Concepts)** page on the ROS wiki.

A New Package

First order of business in ROS is typically to create a new package for your subproject; for example, the vision system might be (at least) one package, the localization system might be one, and so on.

Run the following to create an empty package called part2, and build it (see also **the ROS tutorial on creating packages (https://wiki.ros.org/catkin/Tutorials/CreatingPackage)**):

```
cd ~/dd2419_ws/src
catkin_create_pkg part2 rospy geometry_msgs tf2_ros tf std_msgs crazyflie_driver
catkin build
```

```
source ../devel/setup.$(basename $SHELL)
```

The Node Strikes Back

Though not exactly a leisurely tutorial, there is some **reference material for how ROS CMakeLists.txt are written and structured (https://wiki.ros.org/catkin/CMakeLists.txt)** .

Create a new node called navgoal in your package,

```
roscd part2
mkdir -p scripts
touch scripts/navgoal
chmod +x scripts/navgoal
nano scripts/navgoal
```

You do not have to use `nano`, but it's a pretty decent newbie editor. Then, put the following stub into the file:

```
#!/usr/bin/env python

import rospy
from geometry_msgs.msg import PoseStamped

def goal_callback(msg):
    rospy.loginfo('New goal received:\n%s', msg)

rospy.init_node('navgoal')
sub_goal = rospy.Subscriber('/move_base_simple/goal', PoseStamped, goal_callback)

if __name__ == '__main__':
    rospy.spin()
```

Most of it should be fairly self-explanatory: a callback function that simply prints what is published on the topic `/move_base_simple/goal` is defined (more on this in the next section); the node is registered with its node name to the master; a subscriber is defined; and finally, the node is runs (spins) indefinitely.

You should be able to start your node with `rosrun part2 navgoal`, and also include it in launch files using `<node>` tags. [Note: we do not recommend installing your Python programs using `CMakeLists.txt` due to how the ROS build system handles these programs. It is easier to just not use that function -- unless you're distributing your software.]

RViz

One of the most important tools in the ROS arsenal is RViz, the robot visualization tool. You can start RViz by running `rviz` in a terminal. RViz might look daunting at first, but by the end of this course, you will be an RViz master. At the top is the toolbar. Typically, you will use the "Interact" tool; it pans, zooms and pivots the camera in the main 3D view. On the left is the "displays" panel; for now, you will only have a grid indicating the ground plane. The displays are, as the name suggests, things that are displayed in the main view. On the right is the camera settings panel, for now it can be left alone. It allows you to do things like following the robot in third person, 2D map projection, orbiting, and more.

You'll notice that on the toolbar there's a tool called "2D Nav Goal." What this does is to publish a single `PoseStamped` message to the topic `/move_base_simple/goal`, which happens to be exactly the topic and message type the node from the previous section is listening for. Click in the 3D view and drag to send a pose to your node, and make sure your node prints out what you expect it to.

You might ask yourself what the point is of RViz when there is Gazebo. Gazebo is a simulator, providing sensory data to the robot, while RViz is a visualization of the robot's understanding of the world. So, in real-world settings, Gazebo is not used, while RViz is.

Publishing

Let us take the goal message from RViz and publish it repeatedly to the position command topic. This will cause the Crazyflie to (try to) fly to that position. Create a new node called `navgoal2` the same way you did earlier in this tutorial.

```
#!/usr/bin/env python

import math
import rospy
from tf.transformations import euler_from_quaternion
from geometry_msgs.msg import PoseStamped
from crazyflie_driver.msg import Position

# Current goal (global state)
goal = None

def goal_callback(msg):
    global goal

    # RViz's "2D Nav Goal" publishes z=0, so add some altitude if needed.
    if msg.pose.position.z == 0.0:
        msg.pose.position.z = 0.4

    rospy.loginfo('New goal set:\n%s', msg)
    goal = msg

def publish_cmd(goal):
    cmd = Position()

    cmd.header.stamp = rospy.Time.now()
    cmd.header.frame_id = goal.header.frame_id

    cmd.x = goal.pose.position.x
    cmd.y = goal.pose.position.y
    cmd.z = goal.pose.position.z

    roll, pitch, yaw = euler_from_quaternion((goal.pose.orientation.x,
                                              goal.pose.orientation.y,
                                              goal.pose.orientation.z,
                                              goal.pose.orientation.w))

    cmd.yaw = math.degrees(yaw)

    pub_cmd.publish(cmd)

rospy.init_node('navgoal2')
sub_goal = rospy.Subscriber('/move_base_simple/goal', PoseStamped, goal_callback)
pub_cmd  = rospy.Publisher('/cf1/cmd_position', Position, queue_size=2)

def main():
    rate = rospy.Rate(10)  # Hz
    while not rospy.is_shutdown():
        if goal:
            publish_cmd(goal)
        rate.sleep()

if __name__ == '__main__':
    main()
```

Phew, what a piece of work! Now, if you set a navigation goal in RViz, your (simulated) Crazyflie should be taking off. First, of course, you need to start your simulation as in **Part 1 of flight camp (https://kth.instructure.com/courses/17743/pages/flight-camp-part-1)** :

```
roslaunch simulation simulation.launch
```

One big problem remains -- in what coordinate system are these goals given, and how do we deal with different coordinate systems? It would be useful to be able to say "move forward X meter", or "go to map location Y."

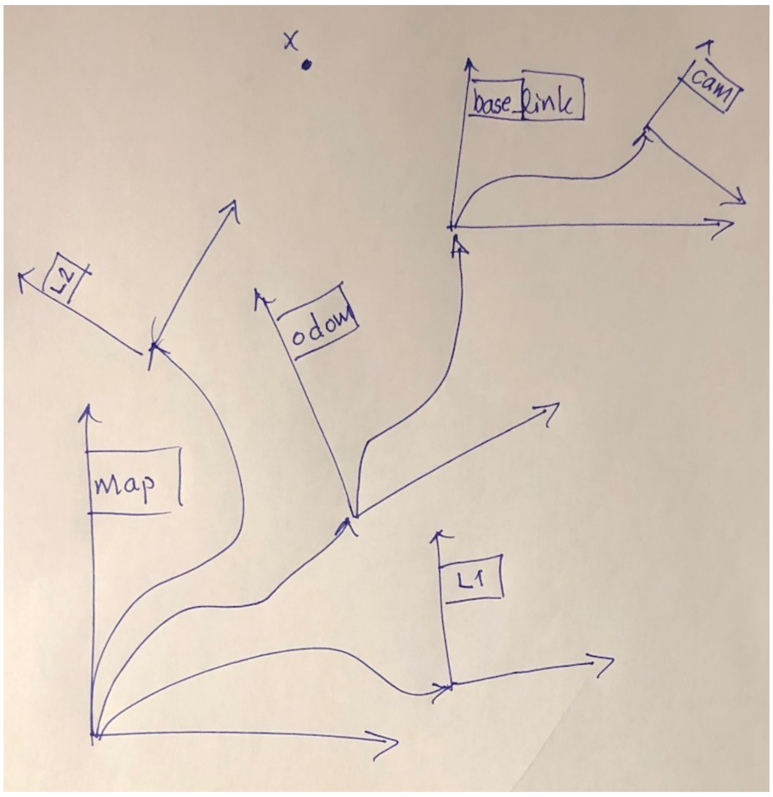
[Intermission] Now would be a good time to take a break, have some fresh air, look at trees and the birds and what they're up to.

Transforms and Frames -- TF

Let's first take a step back and look at how to deal with coordinate systems in ROS. A coordinate system is defined by a frame of reference. You might for example say that your computer is 50 cm from the left edge of your table, and 40 cm from the bottom edge. This coordinate system uses the table's lower left corner as its frame of reference (its origin), but clearly, you could choose any arbitrary reference frame. Perhaps it would make more sense to locate your laptop with respect to the north-east corner of the room you're in. If you then know where the table is in the room, it should be trivial to calculate the position of your laptop in the room -- this is exactly what TF does for you. A more concretely robotics-related application is available in section 1 of **http://wiki.ros.org/navigation/Tutorials/RobotSetup/TF (http://wiki.ros.org/navigation/Tutorials/RobotSetup/TF#Transform_Configuration)** . More exact terminology can be found on the ROS wiki's **TF terminology page (http://wiki.ros.org/tf2/Terminology)** .

In mobile robotics there are three important *frames* that (almost) always are present when working with ROS: the **map** frame, considered the "absolute" coordinate system in that it doesn't change over time; the **odom** frame, for **odom**etry frame, whose origin typically is where the robot was powered on; and the **base_link** frame, which is the robot frame -- in other words, the robot is always at the origin of the **base_link** frame. **REP 105 (http://www.ros.org/reps/rep-0105.html)** defines these, and also gives some naming conventions and semantics, as well as some additional common frames.

The picture below illustrates the relationship between the three frames mentioned above and some additional ones. A position x=(x, y, z) or a pose (x, y, z and three rotations about the body axes) can be expressed in any of the frames. However, one of them is usually more natural than the other. For example, your laptop is easier to locate with respect to the table than the room in the example above. In the image below, the location of the landmarks **L1** and **L2** are easier to express in the map frame, whereas the position of the **camera_link** is defined with respect to **base_link** (i.e. relative to the robot.) We can see from the graph that in order to know where the camera is in the map frame we also need to know where the robot is in the **odom** frame and the relation between the **odom** frame and the **map** frame. This requires a localization system, which estimates the pose of the robot in the map frame and therefore calculates the transform between map and odometry. We will see later how TF allows us to seamlessly move from frame to frame and thus make it possible to, for example, express the location of the landmarks in the **base_link** frame.



Question: What can you say about the transform from **map** and **odom** for a robot that has perfect odometry?

Practical TF

Enough reading. Time to get hands on. Let us implement a node that publishes TF frames at the location of some ArUco markers that you looked at in Part 1 of flight camp (https://kth.instructure.com/courses/17743/pages/flight-camp-part-1) . Use the following tutorial (http://wiki.ros.org/tf2/Tutorials/Writing%20a%20tf2%20static%20broadcaster%20%28C%2B%2B%29) (also available using C++ (http://wiki.ros.org/tf2/Tutorials/Writing%20a%20tf2%20static%20broadcaster%20%28C%2B%2B%29)) as a guide for the general structure. Notice that since the ArUco markers don't move, we publish static tfs for them. Create a new node called `displaymapmarkers` as before.

For this course, we will give you the position of the ArUco markers, obstacles and so on as a JSON file, download example_world.json (https://kth.instructure.com/courses/17743/files/2136133/download?wrap=1) (https://kth.instructure.com/courses/17743/files/2136133/download?wrap=1) , but don't spend too much time on the file contents; we will explain them later on in the course.

Below is the source code for a node that reads the JSON file and publishes a transforms between `map` and `aruco/markerX` for each ArUco marker, where X is the ID of the ArUco marker. Paste this into the node file displaymapmarkers

```
#!/usr/bin/env python

import sys
import math
import json

import rospy
import tf2_ros
from tf.transformations import quaternion_from_euler
from geometry_msgs.msg import TransformStamped, Vector3

def transform_from_marker(m):
    t = TransformStamped()
    t.header.frame_id = 'map'
    t.child_frame_id = 'aruco/marker' + str(m['id'])
    t.transform.translation = Vector3(*m['pose']['position'])
    roll, pitch, yaw = m['pose']['orientation']
    (t.transform.rotation.x,
     t.transform.rotation.y,
     t.transform.rotation.z,
     t.transform.rotation.w) = quaternion_from_euler(math.radians(roll),
                                                     math.radians(pitch),
                                                     math.radians(yaw))
    return t

def main(argv=sys.argv):
    # Let ROS filter through the arguments
    args = rospy.myargv(argv=argv)

    # Load world JSON
    with open(args[1], 'rb') as f:
        world = json.load(f)

    # Create a transform for each marker
    transforms = [transform_from_marker(m) for m in world['markers']]

    # Publish these transforms statically forever
    rospy.init_node('displaymapmarkers')
    broadcaster = tf2_ros.StaticTransformBroadcaster()
    broadcaster.sendTransform(transforms)
    rospy.spin()

if __name__ == "__main__":
    main()
```

Familiarize yourself with the code above. The reason it uses *static* transforms is that TF tries to match timestamps between frames, "going back in time" so to speak when necessary. The idea is that since the robot's sensors and estimation systems don't necessarily run in lock-step, it's necessary to fit the timings together. For our case, the transforms never change, and static transforms are just that: a signal that this transform does not change and is valid for any requested time. Now, run this node with the JSON file as the sole argument:

```
rosrun part2 displaymapmarkers ~/dd2419_ws/src/course_packages/dd2419_resources/worlds_json/tutorial_
1.world.json
```

Open RViz, look in the displays panel on the left, click Add in the bottom of the panel, select "TF" and click Ok. This display visualizes the TF tree, and should show you where the ArUco markers are with respect to the map frame (and eventual other frames.)

Also take a look at the TF tree using the tool rqt which gives you a more graphical view. Run

```
rqt &
```

in a terminal and then choose rqt_tf_tree (http://wiki.ros.org/rqt_tf_tree) from Plugins tab.

Question: Do you have one tree or several sub-trees? Explain what you see! What is missing?

Odometry and Map Frames

For true autonomous operation, the robot needs to be able to localize itself, as mentioned before. That is to say, starting from somewhere in the map, the robot should be able to find out where it actually is in the map frame using the ArUco markers that it has found given that the robot knows a) where the markers are in the **map** frame, b) where the detected markers are in the **camera_link** frame, and c) where the camera is relative to the robot's **base_link**. Then, the transform between **map** and **odom** frames is simply the difference between current pose in **map** and **odom** frame.

A somewhat less sophisticated approach that does not offer autonomous operation is to put the robot at some knows starting location, so that the transform between **map** and **odom** is known in advance. This is obviously a brittle approach and one that will only work for a short time until the transform between map and odom has changed too much due to drift, but it will suffice for demonstrating how to work with the different frames. In our case, the robot starts at (0, 0, 0), with a yaw of 0 radians, in the simulator, and the robot's internal estimator starts at position (0, 0, 0), with a yaw of 0 radians. Thus, the transform between the two frames is (0, 0, 0) and yaw 0. You can publish this transform statically using the `static_transform_publisher` too directly on the command line. Open a terminal and run

```
rosrun tf2_ros static_transform_publisher 0 0 0 0 0 0 map cf1/odom
```

With this transform in place, it should now be possible to visualize the robot's odometric pose in the map frame. Add a new display, and select "By Topic", then find `/cf1/pose`.

Question: Inspect the TF tree again and compare to what you had before. Is there enough information to move between any frame to any other? If not what is missing?

Question: What would happen if the robot didn't start with its X axis aligned to the map's X axis?

Transforming Objects in the World

The `navgoal2` node we previously wrote has an issue: it sends the position setpoint in whatever frame it was received in, which is incorrect -- for example, if you would send a setpoint of 0, 0, 0 in the `aruco/marker1` frame, that would mean "go to marker 1," but the robot would interpret it as "go to odometric position 0, 0, 0." Clearly, we need to recalculate, or *transform*, the goal to something equivalent in the **odom** frame.

First, we need a buffer to store accumulated TF transforms:

```
tf_buf = tf2_ros.Buffer()
```

Second, something that listens to the `/tf` and `/tf_static` topics in ROS and stuffs them into our buffer:

```
tf_listener = tf2_ros.TransformListener(tf_buf)
```

Now we're ready to use our buffer to transform our goal to the odometric frame:

```
goal_odom = tf_buf.transform(goal, 'cf1/odom')
```

If TF cannot find a chain of transforms to get to the target frame, it will raise an exception. It is therefore often a good idea to first test if the transform is available:

```
if not tf_buf.can_transform(goal.header.frame_id, 'cf1/odom', goal.header.stamp):
    print('cant do that, Dave')
```

Putting this all together, into a new node called `navgoal3`, you should have something like:

```
#!/usr/bin/env python

import math
import rospy
import tf2_ros
import tf2_geometry_msgs
from tf.transformations import euler_from_quaternion
from geometry_msgs.msg import PoseStamped
from crazyflie_driver.msg import Position

# Current goal (global state)
goal = None

def goal_callback(msg):
    global goal

    # RViz's "2D Nav Goal" publishes z=0, so add some altitude if needed.
    if msg.pose.position.z == 0.0:
        msg.pose.position.z = 0.4

    rospy.loginfo('New goal set:\n%s', msg)
    goal = msg

def publish_cmd(goal):
    # Need to tell TF that the goal was just generated
    goal.header.stamp = rospy.Time.now()

    if not tf_buf.can_transform(goal.header.frame_id, 'cf1/odom', goal.header.stamp):
        rospy.logwarn_throttle(5.0, 'No transform from %s to cf1/odom' % goal.header.frame_id)
        return

    goal_odom = tf_buf.transform(goal, 'cf1/odom')

    cmd = Position()

    cmd.header.stamp = rospy.Time.now()
    cmd.header.frame_id = goal_odom.header.frame_id

    cmd.x = goal_odom.pose.position.x
    cmd.y = goal_odom.pose.position.y
    cmd.z = goal_odom.pose.position.z

    roll, pitch, yaw = euler_from_quaternion((goal_odom.pose.orientation.x,
                                              goal_odom.pose.orientation.y,
                                              goal_odom.pose.orientation.z,
                                              goal_odom.pose.orientation.w))
    cmd.yaw = math.degrees(yaw)

    pub_cmd.publish(cmd)


rospy.init_node('navgoal3')
sub_goal = rospy.Subscriber('/move_base_simple/goal', PoseStamped, goal_callback)
pub_cmd  = rospy.Publisher('/cf1/cmd_position', Position, queue_size=2)
tf_buf   = tf2_ros.Buffer()
tf_lstn  = tf2_ros.TransformListener(tf_buf)

def main():
    rate = rospy.Rate(10)  # Hz
    while not rospy.is_shutdown():
        if goal:
            publish_cmd(goal)
        rate.sleep()

if __name__ == '__main__':
    main()
```

Please take a moment to look at the added lines (in **bold**). In particular, importing tf2_geometry_msgs is essential though it might not seem like it. Now you can publish goals in any frame you prefer!

Question: The TF transform could be done in the goal callback instead, arguably making the code easier to read. What is the problem with that solution?