POLITECNICO
MILANO 1863

First Robotics Project

Politecnico di Milano

Lorenzo Gadolini, 10522690 Davide Lorenzi, 10502570

Contents

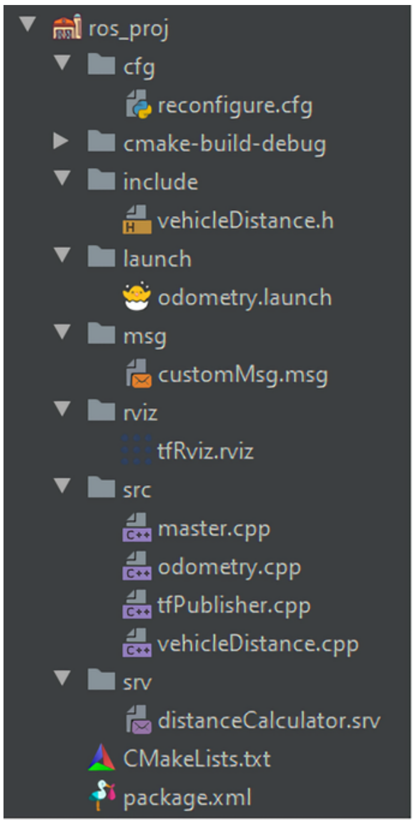
1 Description

The first part of the Robotics Project 2020 consists of computing the distance of two vehicles on the Monza ENI circuit and return the status (SAFE, UNSAFE, CRASH) based on their relative distance. GPS data are coming from a recorded bag that contains 2 topics, one for the car and one for the obstacle. Data are converted to ENU coordinates before the computation.

2 Project structure

We organized the project in the following way:

- The `project` folder contains all the required files for building and executing the simulations.
- The `cfg` folder contains the config file for the dynamic reconfigure.
- The `include` folder contains the header file of the service responsible of calculating distance.
- The `launch` folder contains a .launch file, responsible of simulating a Terminal call to start each node.
- The `msg` folder contains a .msg file. This file is the prototype of the custom message that contains the distance between the vehicles and the relative status tag.
- The `src` folder contains all the .cpp files that implement all the nodes' behavior.
- The `srv` folder contains the .srv file that describes the structure of the service responsible of calculating the distance. The content of this file is a list of data types.
- The `rviz` folder contains the configure to visualize `tf` in rviz that will be exchanged during each service request and response.



```
▼ ros_proj
  ▼ cfg
      reconfigure.cfg
  ▶ cmake-build-debug
  ▼ include
      vehicleDistance.h
  ▼ launch
      odometry.launch
  ▼ msg
      customMsg.msg
  ▼ rviz
      tfRviz.rviz
  ▼ src
      master.cpp
      odometry.cpp
      tfPublisher.cpp
      vehicleDistance.cpp
  ▼ srv
      distanceCalculator.srv
    CMakeLists.txt
    package.xml
```

3 How to compile the simulation

1. After downloading the `src` folder and placing it in an empty directory, you need to run the command `"catkin_make"` in a terminal. This will build the workspace, and compile the .cpp files.
2. To install the custom message and the service, you need to run in the terminal `"catkin_make install"`.

   This command will compile the `.msg` and `.srv` files into cpp code and headers and will enable ROS to use them.
3. After building the project in a new workspace, ROS needs to be able to access all the project files, and to do that, you also need to run `"source devel/setup.bash"` in the terminal.
4. Now ROS can finally start our simulation.

4 How to start the simulation

If everything went according to plans, now is time to invoke the ROS core, and start our executables.

1. To start the ROS middleware, run `"roscore"`in a separate terminal, without ever closing it. Then, you need to launch all the nodes in the project.
2. From the root workspace folder "cd" into the launch folder (or open it in the file manager and then right-click, Open in Terminal) and type`"roslaunch odometry.launch"`*.

   After this command, in the terminal should appear six nodes with their relative name.
3. Now the only thing left to do is starting a bag.

5 Architecture and Development

Our project is designed to use six nodes for the simulation.

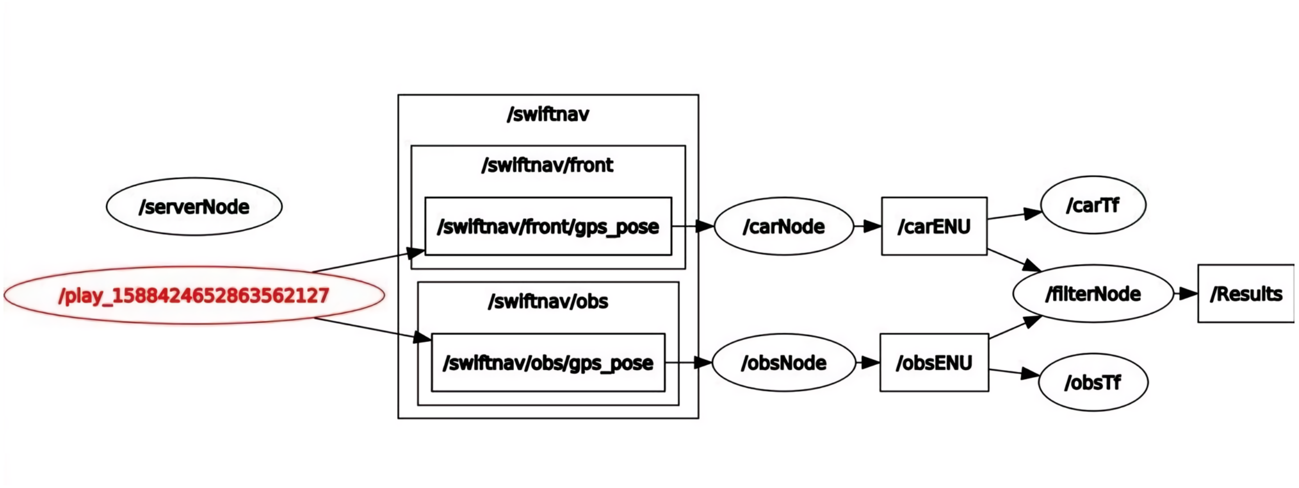
- Two **encoder** nodes are used to read the data coming from the bag's topics. The task of each node is to read from the bag a set of geographical coordinates, encoding them into ENU and publish the new data onto a new topic, wrapped in a `nav_msgs/Odometry` message.
- One **master** node subscribes both the two new ENU topics with a message filter. After subscribing those two topics with a message filter, that implements an approximate time filtering policy, this node pushes all the valid data onto a service call. The data to be flagged as valid needs to have all three of its coordinates not equal to 0. A message like (0,0,0) represents a satellite fix error, and will not be placed in a service call.
- This master node also manages the **dynamic reconfigure**'s parameters. We opted for a solution with two variable thresholds that set interval in which the car is in a `SAFE, UNSAFE, CRASH` status.
- The **service** node listens for all valid requests, and all this node has to do is to compute the distance. The distance is computed using the Euclidean distance formula, and returned to the caller node.
- For each valid distance the master node then proceeds to create a custom message containing two values, the distance and a status flag, and subsequently proceeds to publish them onto a results' dedicated topic.
- The last two nodes are the `tf` nodes, they subscribe separately to the previously created `nav_msgs/Odometry` topics, and their task is to compute the `tf` messages.

*We left uncommented the two lines that start both rviz and rqt reconfigure, if not needed comment them out.

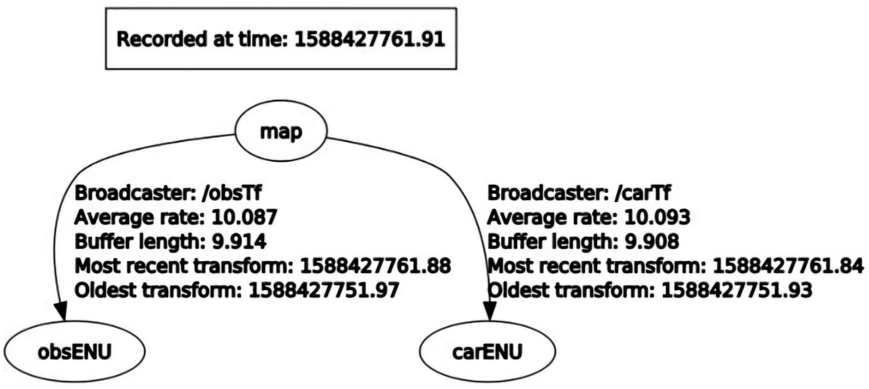
6 How to visualize each single process

If everything went well, and the simulation started, now there is a number of processes running in background. ROS provides a series of tools to visualize the general structure of the simulation.
To see the graph of nodes, services and topics, run in a terminal the command `"rqt_graph"` while to see the `tf frame_id` and `child_frame_id`, run `"rosrun rqt_tf_tree rqt_tf_tree"`.
Those two commands will provide a graphical overview of the processes.

Rqt Graph



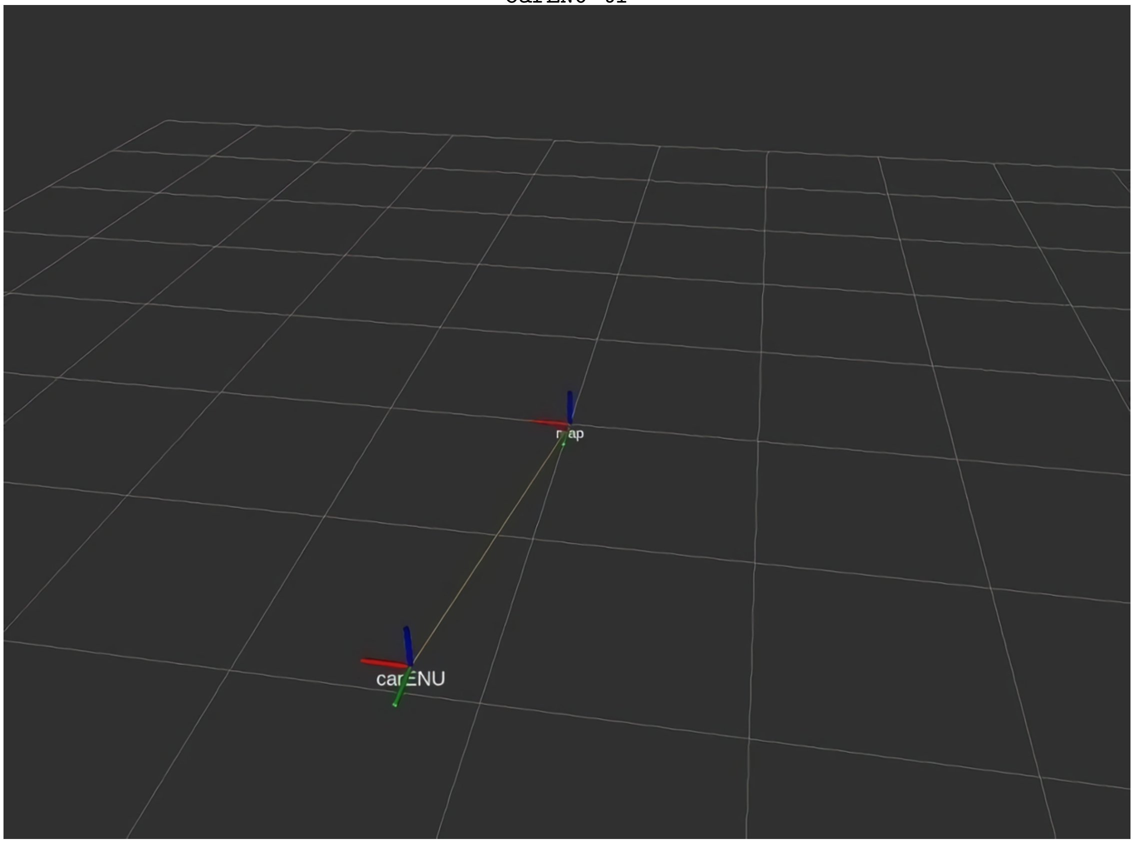
Rqt Tf Tree Graph


Recorded at time: 1588427761.91
map
Broadcaster: /obsTf
Average rate: 10.087
Buffer length: 9.914
Most recent transform: 1588427761.88
Oldest transform: 1588427751.97
Broadcaster: /carTf
Average rate: 10.093
Buffer length: 9.908
Most recent transform: 1588427761.84
Oldest transform: 1588427751.93
obsENU
carENU

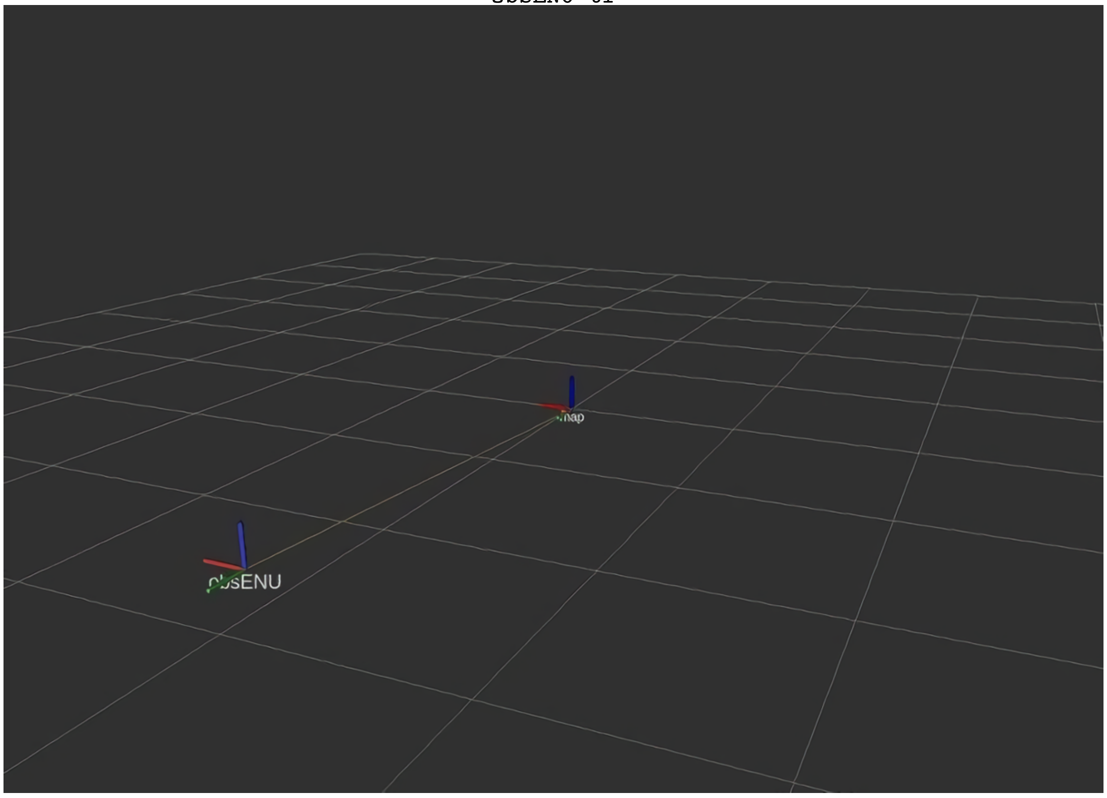
Broadcasting the `/tf` messages means that `rviz` can be used to see the movements of the vehicles that are being simulated with the bag's data.
To be able to see them, you need to start a rviz window with `"rosrun rviz rviz"` while the simulation is executing. Then you need to set the fixed frame as "map" and include a visualization Tf object. `Rviz` should start plotting two points relative to the fixed frame [†]

[†] In order to be able to see the simulation on rviz, we are currently reducing by a factor of 100 the ENU coordinates inside the two `tf` nodes. All the odometry data outside those nodes are coherent with the ENU encoding.

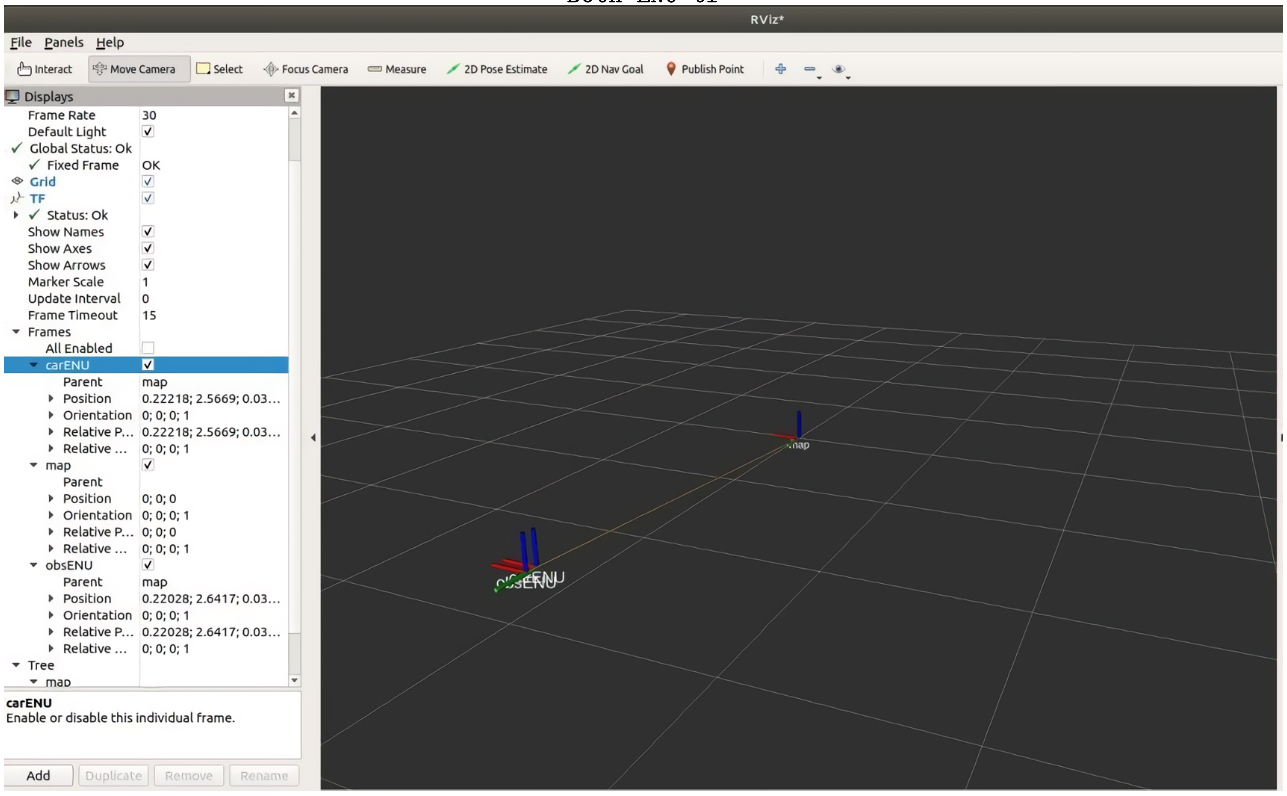
CarENU tf



ObsENU tf



Both ENU tf



```
Frame Rate           30
Default Light        ✓
✓ Global Status: Ok
  ✓ Fixed Frame      OK
Grid                 ✓
TF                   ✓
  ✓ Status: Ok
  Show Names         ✓
  Show Axes          ✓
  Show Arrows        ✓
  Marker Scale       1
  Update Interval    0
  Frame Timeout      15
  Frames
    All Enabled      ☐
    carENU           ✓
      Parent         map
      Position       0.22218; 2.5669; 0.03...
      Orientation    0; 0; 0; 1
      Relative P...  0.22218; 2.5669; 0.03...
      Relative ...   0; 0; 0; 1
    map              ✓
      Parent
      Position       0; 0; 0
      Orientation    0; 0; 0; 1
      Relative P...  0; 0; 0
      Relative ...   0; 0; 0; 1
    obsENU           ✓
      Parent         map
      Position       0.22028; 2.6417; 0.03...
      Orientation    0; 0; 0; 1
      Relative P...  0.22028; 2.6417; 0.03...
      Relative ...   0; 0; 0; 1
  Tree
    map

carENU
Enable or disable this individual frame.
```

7 Conclusions

Our project was designed as modular as possible. There are 4 .cpp files that implement the nodes. Two of those, once built are executed two times (the `tf` executable and the encoder). This modularity helped under the maintainability and debugging perspective, as its quite easy to add features and debug them since they have their own executable.

In regards of testing, we found out after multiple executions that the first ~ 100 seconds of the bag aren't useful to the simulation, because the satellite data from the obstacle is not recorded. This also happens near the last minute or so of the bag. We avoided testing those two particular time frames, as the message filter would discard all the packets, so we skipped them launching `"rosbag play -s 100 project.bag"`.

The filtering policy gave us quite some trouble, as it seems that varying the `queue_size` param introduce a remarkable error in the distance computation. This is due to the fact that the ApproximateTimeFilter may be associating messages that are quite far from each other temporally, so the distance is computed using the wrong coordinates. For this situation where real-time data is fundamental, all the `queue_size` parameters are set to 1.

It is also interesting in our opinion to point out that this design is strongly independent from the situation, hence it would be possible to repurpose this software to compute distance between two sources, as long as the input messages belong to a navsat topic. The only needed modification is to change the input's topic name. However, we would also like to point out that for the sake of simplicity we hardcodded a couple of arguments inside one or two functions, mainly names of topics that are used from the nodes to interact, so any change except the input's topic in the launcher's args fields may result in a broken or not working simulation.