Furbot ROS interface

Ruslan Aminev, Luigi Secondo

EMARO
University of Genoa

Abstract

Furbot is a vehicle made in University of Genova. It is designed to be manually controlled by an operator, but it has an interface to control it via special protocol. It will be more useful make Furbot autonomous at some level.

Our project is part of a platform project, this means that more than one group works to the same robot in order to make a coordinate work. Our main goal was to develop a ROS interface for Furbot vehicle, in order to make it able to be controllable via ROS.

Introduction

Furbot is the acronym of Freight Urban RoBOTic and it is a vehicle designed in University of Genova for sustainable freight transport in urban areas. It has two slots to carry two standart size pallets, and two forks to load and unload them (Figure 1).

Figure 1: Furbot right side view.

Right now Furbot is only controllable manually by a human driver onboard via joystick and pedals, and the project is supposed to allow driving the Furbot via an external computer in order to implement self driving capabilities. In other words the goal of the project is to implement software interface to control the vehicle.

In robotics research ROS is the most spread solution to bring together different software components. There already exist a lot of high level control, and decision making algorithms for car-like robots. So it was obvious to use ROS for our platform project. The goal of our group project was to implement a ROS interface in order to control Furbot at a low level.

Requirements and preliminar phase

From general ROS best practices we learned that a ROS interface for Furbot should be able to subscribe to a $geometry\_msgs/Twist$ message, that expresses velocity in free space decomposed into its linear and angular parts, and be able to publish Odometry navigation messages $nav\_msgs/Odometry$ and to broadcast transformation matrices tf. In addition to that, it was necessary to make available for the user all the telemetry data of the Furbot.

Communication in Furbot is provided via UDP on which a specific protocol has been developed. The protocol is designed in order to provide information of Furbot's systems internal state and to transmit control inputs from a computer. According to the specification provided, the UDP interface operates at frequency of 100 Hz.

In particular Furbot's communication protocol provides information about battery state, using data computed by the Battery management system

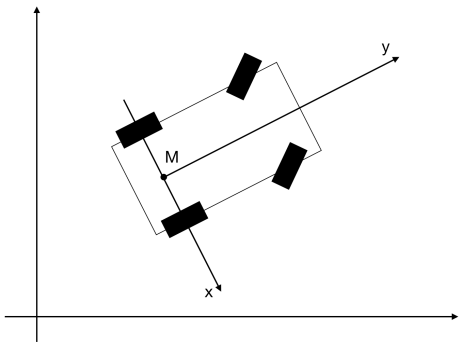

Figure 2: Robot frame.

(BMS), a lot of information about traction data, such as velocities, and steering angles. Also, all the information about hydraulic systems is provided so it's possible to monitor states of vehicle's suspension and the forks used to move pallets. Regarding control inputs to be sent to the Furbot, it is possible to set the desired steering angle of the front wheels, the throttle value and the braking intensity.

During the preliminary phase of the project, it was necessary to extract all the information about Furbot's parameters, such as wheels distances, wheels radius, etc. from the 3D model of the vehicle designed with software CREO Parametric 3D.

In order to make the work of three groups of the platform project coherent, it was necessary to define some common conventions and rules. The most important standard that has been defined is the position of the robot frame following the general convention of the car-like robots, setting the origin of the frame in the middle of the axis connecting the two rear wheels and orienting the axes as shown in Figure 2.

Also we agreed with other groups to use ROS Kinetic for our platform project.

Study existing solutions

During preliminary phase we studied many existing solutions to implement a ROS interface for controlling a vehicle; in particular we gave more attention to solutions developed by Autoware, Dataspeed Inc., Starline and Ecole Centrale Nantes.

Autoware, (1), is a ROS-based open-source software, enabling self-driving mobility to be deployed in open city areas. It provides a strong platform for products and services, on which it is possible to select cameras, LiDAR, GNSS, or IMU as underlying sensors, and run localization and object detection modules using 3D maps, followed prediction, planning, and control modules to actuate by-ware vehicles.

Dataspeed Inc. (2) is a company that distributes a kit, called ADAS Kit, to provide a complete hardware and software solution that allows seamless control of throttle, brake, steering, and shifting in order to assist testing sensors for autonomous vehicles applications.

StarLine (3) is Russian company which main specialization is creating car alarm systems. There is research project called "StarLine smart vehicle". It's purpose is to understand what kind of hardware and software, that brings different autonomous capabilities, company could produce and sell to automakers. They created ROS interface for Skoda Suberb sedan.

After an evaluation of the solutions above, we concluded that it would have been very difficult to try to transfer them to our project, because each of them is very specific, and heavily depends on the vehicle being used. Since the Furbot is also very unique, we thought that it would be easier to work on the project starting from scratch. For this reason we received a great help from Eng. Salvador Dominguez in Ecole Centrale Nantes, where he explained us what was the approach used for their implementation of ROS interface on Renault Zoe. The implementation of the interface will be explained in details in the following chapter.

Software architecture

Starting from overall understanding of how ROS interface for Furbot should be organized and using examples which we have found, we created high level architecture as shown in scheme on the Figure 3. Following the requirements ROS interface should accept as input reference velocities, it

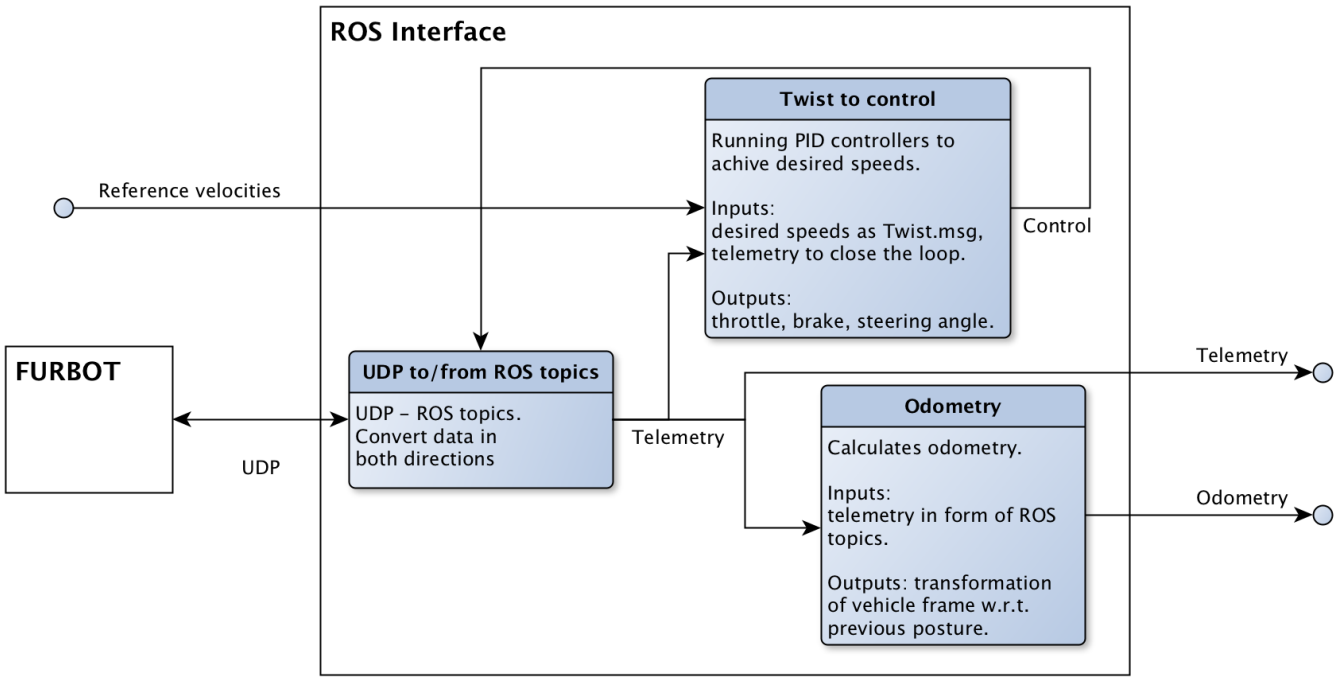

Figure 3: High level software architecture.

should output all the telemetry which is provided by UDP protocol, also it should calculate odometry and publish it in standard way. On Figure 3 there are blocks which perform each of the tasks mentioned above. Also there is a core block for the interface which performs all conversions from UDP to ROS messages and backward.

UDP-ROS interface

The last block is presented in more details on Figure 4, it is combined by two nodes: one for UDP to ROS messages conversion and one for backward conversion. The first one should always listen on UDP and publish telemetry on ROS network. In spite of known frequency of Furbot's UDP messages it is not safe to process them in the same thread with ROS publishers. The better solution is to run separate thread, which continuously listens on UDP port for incoming packages. This approach allows to robustly process all UDP messages as it is not scheduled via ROS utils which are used to maintain publishers frequencies. However, this approach requires to use shared memory for the threads. So the main thread, which con-

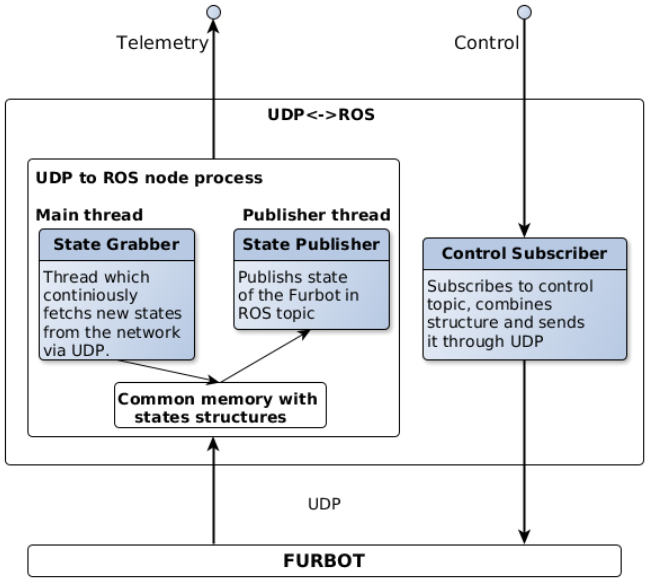

Figure 4: UDP - ROS interface architecture.

tinuously listens on UDP, parses UDP frames to grab telemetry data and puts it in special structures in shared memory. The second thread is responsible for publishing everything on ROS net-

work. It runs a publisher for each Furbot's system according to protocol, each publisher reads data from corresponding structure in shared memory, creates custom message for each subsystem, and publishes it with defined frequency. We decided to use 100 Hz as Furbot uses the same frequency for UDP communication. As this node starts, it checks all status flags for each subsystem, when all flags confirm that Furbot is in working condition, the second thread is started and detached to run publishing on ROS network. After this the node works in the manner described above.

The second node responsible for communication with Furbot is more simple. It just subscribes to topic with custom control signals messages $furbot\_msgs/ControlSignals$, the subscriber callback packs control signals in special UDP frames according to the protocol, and sends it to the vehicle.

Twist to control

According to the established requirements Furbot's ROS interface should accept control in form of $geometry\_msgs/Twist$ message, which specifies desired velocities. However, Furbot UDP protocol is designed in the way that only allows to control Steering angle, Throttle, and Brake. To produce this signals it's required to run PID loops. To close feedback loops we use custom messages $furbot\_msgs/SteeringData$ and $furbot\_msgs/TractionData$, which are published by first node described in UDP-ROS interface part of this report. This node outputs custom message with control signals $furbot\_msgs/ControlSignals$. Simple scheme on Figure 5 shows messages' relation.

This node receives desired velocities as command signals. For throttle and brake simply current velocities from $furbot\_msgs/TractionData$ message and desired velocities from $geometry\_msgs/Twist$ message could be used in PID control loop. Whereas for steering angle control it's required to convert desired angular velocity to desired steering angle. For this part we used simple

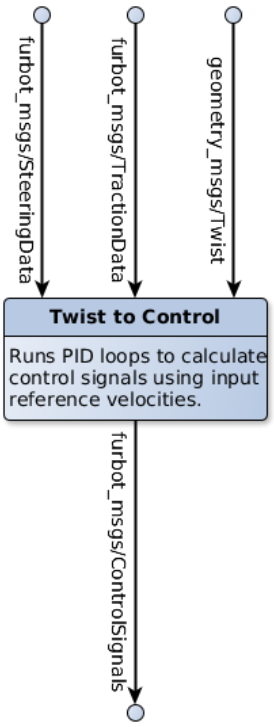

Figure 5: Twist to control signals conversion node.

formulas:

- if command speed is higher than $0.1m/s$

$$\Theta_s = atan(wheelbase * \omega_z / v_y),$$

- else, if vehicle speed is higher than $0.5m/s$

$$\Theta_s = atan(wheelbase * \omega_z / speed),$$

- else

$$\Theta_s = 0.$$

Where ω_z is command angular velocity about z axis of Furbot's frame, v_y is command linear velocity along y axis, and speed is current speed of the Furbot.

The same formulas are used in Dataspeed's solution for ROS interface for Lincoln MKZ sedan (4).

Odometry

The odometry node computes transformation of the robot frame between previous state and current state, based on the data received from a custom message created by us. The message is called $furbot\_msgs/TractionData$ and contains many information about Furbot's traction state. In particular the $TractionData$ custom message is structured as shown in Table 1.

Value type	Name
Header	header
int8	state
int8	mode
int16	speed
int16	vel_l
int16	vel_r
int16	throttle
int16	brake
int32	odo_travel

Table 1: Custom message of Furbot's traction data: $furbot\_msgs/TractionData$.

The algorithm used for odometry is based on the unicycle model and, since it is easier for the computation, makes the computation on the rear wheels velocity. The node, as explained above, subscribes to a custom message in which are specified rear wheels speeds (and other useful values), converts them in linear velocities, computes v and ω and obtains the new position vector of the robot frame using the following formulas:

a) $x_{new} = v \cdot \Delta t \cdot sin(\theta)$

b) $y_{new} = v \cdot \Delta t \cdot cos(\theta)$

c) $\theta_{new} = \omega \cdot \Delta t$

In order to compute transform matrices over ROS, it is necessary to use the tf package (5) that allows to keep track of multiple coordinate frames over time. tf maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc. between any two coordinate frames at any desired point in time.

Once obtained the new position vector of the robot frame, the node has to broadcast the transform over tf and next, it publishes the odometry message over ROS using a standard message $nav\_msgs/Odometry$

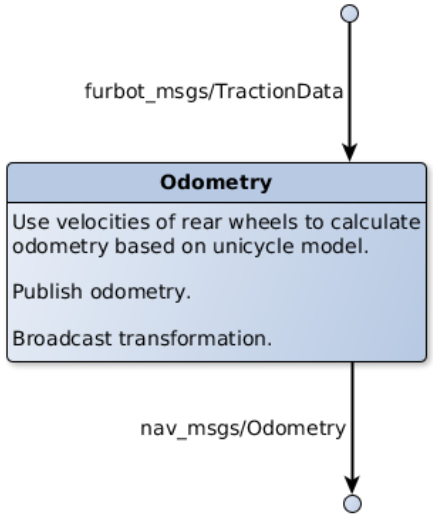

Figure 6: Odometry node.

Implementation

Regarding implementation, as said in the introduction we decided to work on ROS Kinetic distribution and we wrote all the code in C++. Since the project is a low level architecture, it has to be as fast as possible and, for this reason, python could be too slow.

For multithread functionality we used POSIX Threads as it's the most simple solution for Linux. Also it's most common one so everybody could understand our code.

As mentioned before, it was necessary to develop some ROS custom messages in order to use Furbot's data. The messages are $TractionData$, $SteeringData$ and $ControlSignals$. The first one has been explained in details in the Odometry chapter. $SteeringData$ is a custom message in which are contained the current angle of the

front wheels and the desired angle (Table 2). Finally the $ControlSignals$ message is used by the Twist to control node to send control signals to the UDP-ROS interface that will convert the message in UDP frame and send to Furbot; as mentioned before, the message contains a steering angle, the throttle value and brake intensity value (table 3).

Value type	Name
Header	header
int8	state
int16	current_angle
int16	target_angle

Table 2: Custom message of Furbot's steering data: $furbot\_msgs/SteeringData$.

Value type	Name
Header	header
int32	steer
int32	throttle
int16	brake

Table 3: Custom message of Furbot's control signals: $furbot\_msgs/ControlSignals$.

Since the real Furbot was damaged and it was impossible to test our interface directly on it, we had to create some simulation scripts in order to evaluate the efficiency of the project. In particular we created a state sender, which simply sends a set of random (but realistic) values to the UDP-ROS converter node, in order to make it compute UDP signals and we wrote a control receiver, just to have a feedback to check if the whole process works as expected.

Results discussion

This project gave us opportunity to understand how proper ROS interface should be organized in terms of architecture, and how it should be structured in terms of software package. We discovered how to combine together control nodes of different levels, and make them communicating.

What was done

In conclusion, we managed to develop UDP-ROS communication, and the architecture is able to parse all the main data required to control the vehicle and to compute the odometry. The twist to control node and the odometry node are complete and working.

What has to be done

In possible future development, it will be necessary to implement the parsing of the other states in order to make it able to monitor suspension and forks states. Of course, it will be important to tune odometry and control values on the real Furbot, because there could be some mismatching between real values and simulated ones. For the same reason, PID coefficients have to be tuned on the vehicle too. In addition it would be very useful to implement an emergency stop procedure, in order to make the architecture safer.

In conclusion, a good documentation should be provided with the project so that it would be easy to use the architecture and to tune all the parameters in order to improve the performances.

References

[1] Autoware: https://autoware.ai/

[2] Dataspeed: http://dataspeedinc.com/

[3] StarLine: http://www.starline.ru/

[4] Dataspeed Lincoln MKZ ROS interface: https://bitbucket.org/DataspeedInc/dbw_mkz_ros

[5] ROS tf package: http://wiki.ros.org/tf