Acta Polytechnica CTU Proceedings **6**:11–17, 2016
doi:10.14311/APP.2016.6.0011

available online at http://ojs.cvut.cz/ojs/index.php/app

USING ROBOT OPERATING SYSTEM FOR AUTONOMOUS CONTROL OF ROBOTS IN EUROBOT, ERC AND ROBOTOUR COMPETITIONS

Grzegorz Granosik*, Kacper Andrzejczak, Mateusz Kujawinski, Rafal Bonecki, Lukasz Chlebowicz, Błażej Krysztofiak, Konrad Mirecki, Marek Gawryszewski

Lodz University of Technology, Stefanowskiego 18/22, Lodz, Poland

* corresponding author: granosik@p.lodz.pl

Abstract. This paper presents application of the Navigation Stack available in Robot Operating System as a basis for the autonomous control of the mobile robots developed for a few different robot competitions. We present three case studies.

Keywords: ROS, navigation, robot, ERC, Eurobot.

1. Introduction

Students from the Scientific Association for Robotic Research SKaNeR [1] at the Lodz University of Technology (TUL) have been competing in various international robotic contests (e.g. Eurobot, Robot Challenge, Robotour, Sumo Challenge, European Rover Challenge) for several years. They have been working in separate teams focused on robot construction, electronics and control specific for the contest. This year we have decided to join forces and use Robot Operating System (ROS) as a common software platform to build autonomous mobile robots for various competitions.

ROS has been introduced in Automatic Control and Robotics curricula at TUL a few years ago in a very smooth way [2]: (1) by modeling various robots in our web-based application MyModelRobot in order to learn XML syntax and robot structure used in Gazebo and ROS, and (2) by utilizing rosserial and rosserial_arduino to gather sensory data and control small mobile robots. That way the interest in ROS started to grow and we have learned how to use all advantages of this system effectively, and specifically:

- A multitude of tools and libraries that were created around ROS to rapidly prototype the high and middle level of control system for mobile robots.
- Its modern software architecture to easily connect different applications and devices. We could build systems where most of processes work in parallel and on different machines without building multithreading or networking procedures by ourselves.
- Its "thin" ideology — in some cases programs did not need to be highly integrated with ROS.
- We could reuse many tools created outside this system by adding only small parts bridging them to the ROS. Documentation and support provided by books [3], [4], tutorials online, and ROS forum [5].
- Ease of debugging, visualization and logging by using several tools: rxgraph to see system's state graph, rxplot to plot different variables online, Rviz to visualize the whole robot and its sensors readings, or to archive and replay all data.
- Free and Open Access based on straightforward licenses [6]. We could use many libraries, adapt them to our needs and compile by ourselves.

The most complex problem all SKaNeR teams have been facing is an autonomous navigation of a robot. In the further part of this paper we will present our approach to use ROS to solve this problem in the case of three robots built for different contests: Eurobot (robot Husarz), European Rover Challenge (Raptors rover) and Robotour (Quadron and Raptors rover). These robots have different constructions, different kinematics and control systems. Further description starts with application of navigation stack at the Husarz robot then the specific adaptations to other robots are presented.

2. ROS Navigation Stack

The Husarz system (composed of two robots, shown in Figure 1) was developed by our SKaNeR team for Eurobot competitions. Each edition of Eurobot. has different main topic, in year 2016 it was Beachbot and the tasks symbolized enjoying time on the beach.

- The flags — robots must close doors of a shed to rise the flag.
- The sea fishing — robots must catch plastic fishes from a aquarium with water and put them in a net.
- The sand castle — robots must build a sand castle from several elements (cubes, cylinders, cones).
- The sea shell — shells are symbolized by round discs, and they are in five configurations on the playing area. Robot must take shells to the starting place.

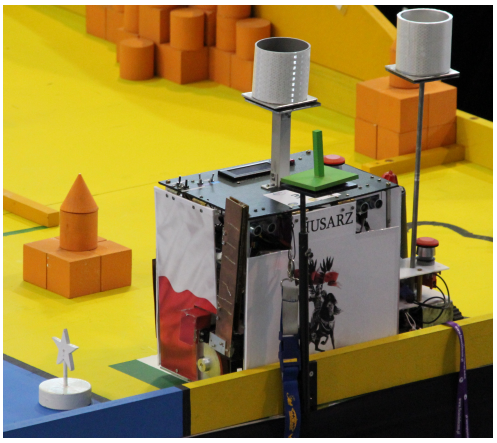

Figure 1. Husarz robots at the Eurobot competitions

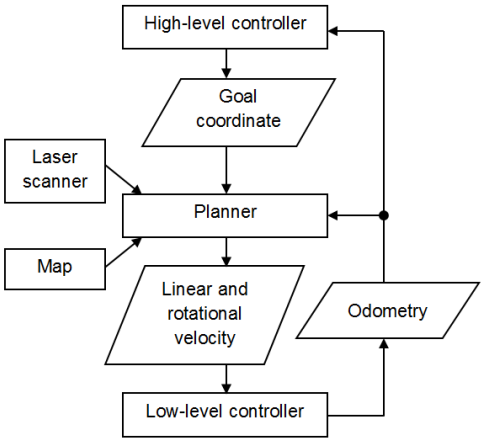

Figure 2. Control sytem of the Husarz robot

- Hide in the shade — this is the funny action, after 90 seconds (the end of a match) robot must rise a beach umbrella to get extra points.

Each team is allowed to have up to two robots on the playing area. Robots must be fully autonomous during a match (no interference with human operator) and avoid crashes with opponents (referee could give negative points for a crash).

We have built two differential drive robots, the bigger one has wheels while the smaller uses tracks. They share the control system schematically shown in Figure 2. We can clearly see that this control system is organized according to the ROS Navigation Stack presented in Figure 3 and in [7].

We used this approach to take advantage of the provided programs (or nodes in ROS nomenclature, see Figure 3): Planner/Navigator (move_base) and the Adaptive Monte Carlo Localization (amcl). On the other hand we had to provide exact structure of the control system and format of all topics (messages being sent between nodes).

The high-level controller knows all tasks (required by competitions) and can choose which robot will perform the specific one. These tasks are provided in the script (in the form of sentences). Every sentence is composed of several stages. The first one always uses the move_base node to set the robot in an appropriate position to run the task. Next steps depend on the type of the task (for example: sending order with a linear velocity directly to the robot to move something on the playing area, or sending order to the manipulators to pick up something, or some combination of such orders). In the application prepared for Eurobot 2016 contest we have specified four global tasks: move pads (1), close doors (2), pick up and bring the sand castle to point area (3) and catch the fish (4). Tasks 1 and 2 could be performed by either robot, while 3 and 4 could be made only by the bigger one. At the beginning of a match, the big robot has got three tasks to do, the small one has one regular task and an extra task to be performed after the end of the match (opening an umbrella after 90 seconds).

Let‘s look closer on the sea fishing task. Always in a first step robot uses move_base planner to get to the point nearby the task execution (close to the aquarium in this case). After the robot gets to the right place (with the right orientation) the main controller sends order to the low-level controller to deploy a manipulator (fishing boom). In the next step the main controller sends the value of linear velocity directly to the low-level controller (which control drivers). The main controller continuously analyzes position read from /odom topic. When the robot reaches the other end of aquarium the main controller sends order to stop, then to return (sends negative value of linear velocity). After that, the controller repeats the above operation one more to go along the wall. On the next run robot goes further to reach a net and leave fishes (appropriate order sent to the manipulator) to score points. Each step has got a time limit. After reaching a time limit before completing a step, the program has to choose either to repeat the step one more time or tag this task failed and proceed to the next task. The main controller counts how many times it sent specific orders. If this value reaches the limit, the program will move to another task.

The high-level controller node publishes the /move_base_simple/goal topic containing information about position and orientation of the target point in the standard format of geometry_msgs/PoseStamped message. This is one of the inputs of the planner — which do all hard work of finding safe path on the playing area and avoiding obstacles — and further publishes /cmd_vel topic containing reference linear and angular velocities for the mobile robot in the format: geometry_msgs/Twist. This topic is subscribed by the node which realizes the low level controller and which is located on the Arduino board controlling motor drivers. The low-level controller also publishes an odometry information.

Planer and navigator hidden in the move_base node has to be configured before use, we have to set robot‘s size, maximum and minimum angular and linear velocities, and accelerations, planner‘s frequency, map

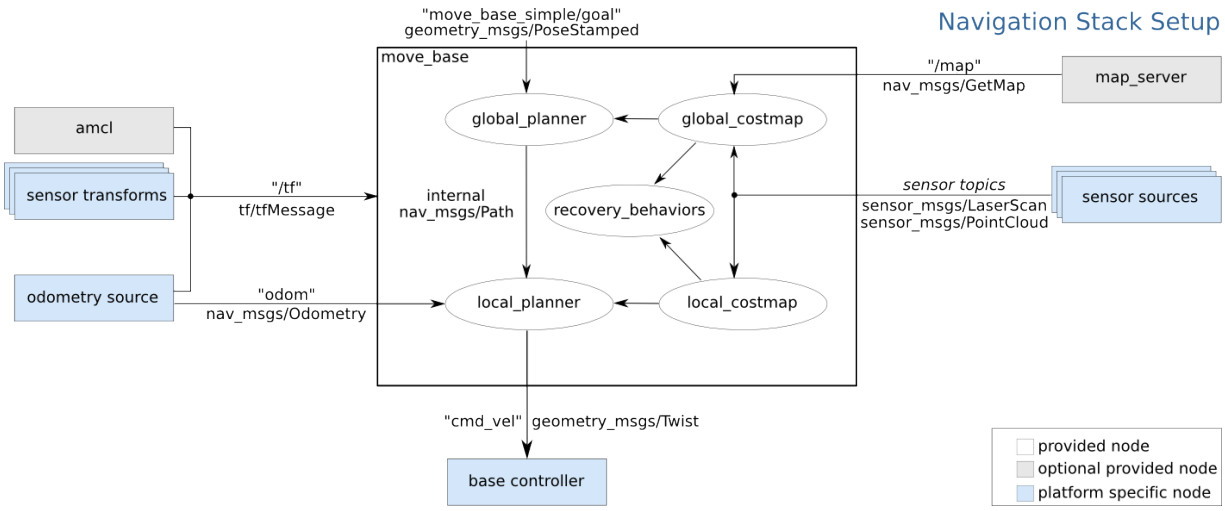

FIGURE 3. Standard ROS Navigation Stack [7].

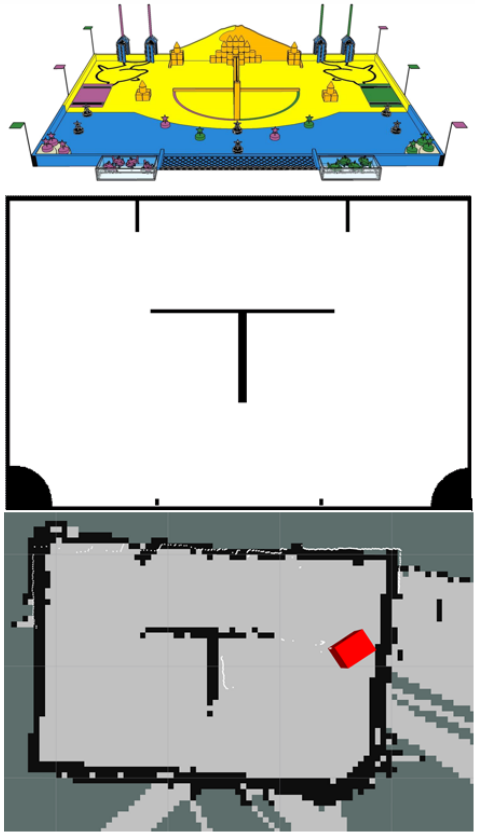

FIGURE 4. Building the navigation map (middle) based on the geometry of playing board (up) of Eurobot competitions, or based on several scans taken on the real board (down)

parameters etc. If the robot has different kinematics one can use another ROS planer leaving the operating principles of the navigation stack unchanged. The most important parameters in case of Eurobot contest were pdist/_scale and gdist_scale:

pdist_scale — the relative importance of sticking to the global path as opposed to getting to the goal.

gdist_scale — the relative importance of getting to the goal rather than sticking to the global path.

By trying different values we have tuned these values pdist_scale = 0.75, and gdist_scale = 1. Our robot do not stick to global path but tries to get to goals in a smooth way. When pdist_scale is larger than gdist_scale the robot drives only trough global path (often stops and sets up orientation) and wastes more time (match lasts only 90 seconds).

The other important information is the map which can be created directly using external sensors (e.g. laser scanner) or prepared in any graphics software. The format of /map topic is also standardized by map.pgm (portable greymap) file. In case of the Eurobot competitions we had to prepare the map based on geometric data describing playing area and a few criterions. The elements on the playing board are divided into three groups:

- fixed (walls),
- movable untouchable (opponent robots),
- movable touchable (playing elements).

Only fixed elements are included in the initial map, as shown in Figure 4 (middle). This map is further used by Simultaneous Localization and Mapping (SLAM) method. Robot localizes itself and creates cost maps in every step of the control. Necessary sensor data are delivered by Hokuyo laser scanner in message sensor_msgs/LaserScan — visible as white dots in Figure 4 (down). These scans together with a laser-based map and transform messages are used by amcl node to produce pose estimates alternative to dead reckoning. Laser scans are also used for obstacle avoidance in the navigator — /local_costmap. Based on the map, current scans and the next target point (provided by the high-level controller), the planner is generating reference path for the robot, as shown in Figure 5.

The high-level controller takes care of the Husarz behavior, it tracks the time of match, monitors communication between nodes and the status of the task.

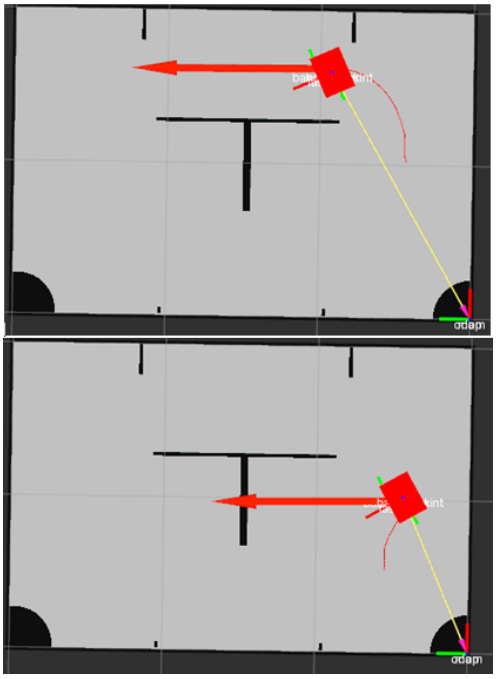

Figure 5. Rviz screens presenting two consecutive steps of planning robot's movements (red line), red arrow shows orientation of the robot required in the step

We have assumed that robot can have four attempts to fulfil each task. These attempts are limited in time, because the match lasts only 90 seconds. After 4 unsuccessful tries the status of the task is switched to "fail" and the robot moves to the next task. Program selects the next task on the basis of distance to move and the remaining time. If one of the robots cannot finish the task it can be sent to the other robot if that robot would be able to run the task. In case of problems with communication the controller will wait until communication is reestablished.

The main controller monitors condition of connections between ROS and robots. We are using a part of standard ROS messages called header. Header includes sequence ID (uint32 format), time stamp (time format) and frame_id (string format). For connection monitoring the most important is time stamp indicating ROS time at the moment when the message was sent. The monitor compares time stamps of incoming messages. When the time stamp does not increase with real time the main controller knows that the connection with robot it lost. In this case main controller goes to stop mode. While the controller is in the stop mode it will not send messages to lower level to avoid situation when the robot loses some orders. In the stop mode the controller continue to check time stamps of incoming messages in order to detect recovery of communication.

2.1. BRIDGING ROS WITH LABVIEW

Another mobile robot requiring mobility planning and autonomous control is Raptors Rover (see Figure 6)

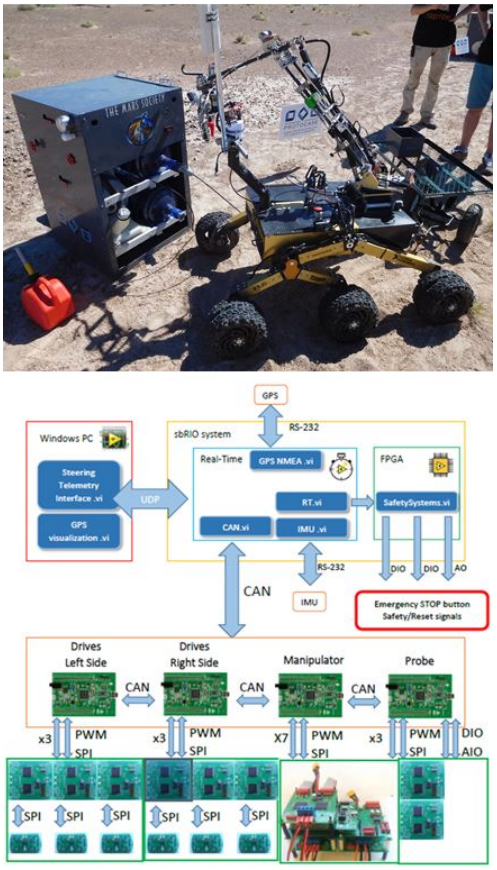

Figure 6. The Raptors rover and its control system

being prepared for European Rover Challenge (ERC). One of the tasks in this contest is navigation — intended to demonstrate rovers and teams ability of approaching locations on the field with limited or without supervision. We have to develop a system which will allow operator to navigate rover without access to visual data such as video streaming, photos and other sensor sources placed on the rover that are presenting visual information.

The Raptors rover was built entirely in house — it is a six wheels platform with well-known rocker-bogie suspension system, wireless communication and several cameras. The robot is also equipped with exchangeable tools: 5DOF manipulator with gripper and drilling sampler. The control system is organized into 4 levels (see Figure 6): the operator's console working on a PC computer and developed in LabVIEW, the onboard main controller based on sbRIO embedded system and developed also in LabVIEW, low-level controllers based on ARM processors with our own software written in C++, and motor drivers. The communication system employs mainly WiFi and CAN networks with local connections based on RS232 and SPI.

Application of the Navigation Stack in this case requires the following modifications the original control system:

- providing coordinates of goal points based on the

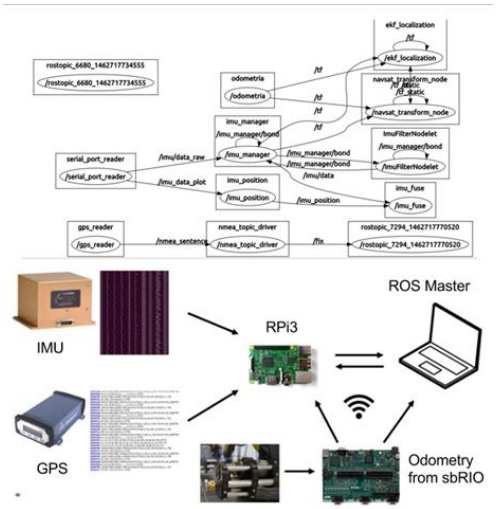

FIGURE 7. Raptors rover navigation system (up) nodes and topics of the ROS structure (down)

topological 3D map of ERC field,

- extending SLAM to 3D or providing alternative source of pose estimation insensitive to traveling on the rough terrain,
- connecting ROS nodes to LabVIEW based control system to seamlessly exchange control and feedback data.

The first requirement can be fulfilled by changing script inside the high-level controller described in the previous chapter. To address the second requirement we have built an independent navigation system based on ROS, as shown in Figure 7. Our system runs on multiple machines with different hardware architectures, file systems and OSs but it is not a problem for ROS. Inertial data is provided by high quality IMU with six degrees of freedom, containing gyroscope and accelerometer. The sensor's output is three axis angular velocity and linear acceleration. Using specialized filters system computes very accurate orientation. GPS module outputs position data with one meter accuracy. These data pass through the Raspberry Pi3 (RPi3) microcontroller and Ethernet to the computer in base station that is ROS master and process the most demanding calculations (this computer can be finally placed on the robot for full autonomy). Wheel odometry is read from the sbRIO board via WiFi connection.

To address the third modification correctly we need to understand that ROS requires all nodes to be created in certain way and in specific programming language (usually Python or C++). However, there are appliances which requires specific programming language or hardware platform, and this leads to problem with incompatibility of such ROS and non — ROS system. This problem is common for various robotics projects, and two bridging methods between ROS and external applications were proposed:

- rosserial — simple wrapper for ROS messages, designed to allow communication between microcontrollers and ROS nodes,
- ros_bridge — dedicated node, which translate ROS messages into standard JSON format, and route them to the right node. Non-ROS application is required to provide support for WebSockets and serialization of JSON messages, as shown in Figure 7.

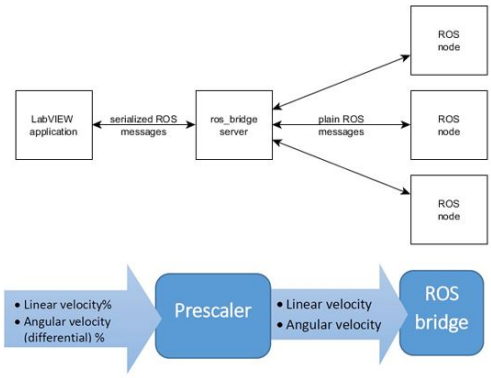

FIGURE 8. ROS bridge functionality

In our application, sbRIO board runs the LabVIEW application, which is not able to talk with ROS nodes. However, LabVIEW includes functions allowing serialization and deserialization of messages in JSON format and is capable to communicate over the network. Ros_bridge package was chosen as a method of communication, because of its flexibility and full support of ROS message format.

Additionally, format of the control data in the Raptors rover is different than required by Navigation Stack. Robot has 6 independently driven wheels, the steer maneuver is obtained by differential control of the wheel speeds of the left and right side of the robot and these speeds are scaled in per cents. Linear velocity is in the range 0-255 (0-100%), the angular velocity is also 0-255 (0-100%), but we do not steer directly the angle, instead we use a difference of control between the left and right side drives of the robot. Due to the further coding of these control signals on the CAN bus (in order to send them to STM modules), we have used the following ranges of values of a linear speed: 127-128 (50%) means stop, 0-127 is reverse, and 128-255 is forward. Similarly, the angular velocity of the robot or the difference between the left and the right side is zero for values of 127-128 (50%), the range 0-127 means turning left, and 128-255 means turning right.

To obtain a uniform data format for input to the ROS module, before sending data through the ROS bridge we have to scale it in an additional vi (section of main program in LabVIEW) which produces a reference linear velocity and angular velocity — see Figure 8

During tests we were using Crossbow IMU800CA which was directly publishing raw data over RS232. In this case we were forced to use a specialized filter,

as mentioned, to parse the data from the raw digital sensor outputs to proper quaternion based format. We have tried two filters from the navigation stack: Madgwick's algorithm and complementary filter. They can be found in imu_tools package. These filters are responsible for inertial data fusion and noise reduction. We have also tested the new IMU unit that is Xsens MTi 1. It has both accelerometer and gyroscope and additional magnetometer on board. Usually we would have used our node that reads data from serial port but more often ROS comes with packages supporting particular hardware eq. Xsens. The package we use is called xsens_driver and contains nodes for gathering data and launch file that even automatically searches for a right serial port and baudrate. For getting GPS data from our Omnistar 3200lr12 we use the excellent nmea_navsat_driver that plugs to the serial port directly and parses NMEA sentences from GPS unit into GPS location form on-the-fly. The final data in a form of digital latitude and longtitude is being sent in sensor_msgs/ NavSatFix via /fix topic. The only thing we had to do here was to create the launch file that indicates which sub-nodes to launch and which serial port and baudrate to choose. Wheel odometry data has longer way to go. It is published from wheel encoders via CAN bus to the sbRio computer and then using rosbridge and JSON format to ROS. In a specialized node it is used to calculate position in respect of rover's kinematics. Having these three data lines, we could use it all over the distributed ROS system eq. plug it to a Kalman filter available in robot_localization node and connect the output topic of estimated odometry to planning structure made by our colleagues from Eurobot team.

2.2. QUADRON ROBOT

We have built Quadron robot (shown in Figure 9) to compete in Robotour contest. It has a carlike kinematic configuration, the base of this vehicle is an electric quad. We have introduced a number of electrical and mechanical modifications, especially in the supporting frame, the drive systems and steering mechanisms, that now can be done with electric motors. For the main drive we have used 500W DC motor, while for the steering the smaller DC gear motor and a coupler system developed in house. Each engine is equipped with a driver, encoders and security features the same as used in Raptors rover.

The robot is further equipped with a number of sensors and modules responsible for security, localization and navigation in the field: navigation module with GPS and IMU (described earlier), SICK LMS291 lidar and ultrasonic sensors facing forward.

The concept of a multilayer control required the use of computers at different levels: ARM microcontroller (STM32 Nucleo) at the lower, and mini size PC at the higher layer, as shown schematically in Figure 10. Entire system is based on ROS. The low level node with the software written in C++ is interfacing motors,

Figure 9. Quadron robot for Robotour contest

encoders, sensors and gamepad. It also receives a higher level commands from the PC, sends appropriate signals to the motors and data from sensors to PC. The board and the PC communicate using rosserial. Basically, Nucleo is connected to the computer using a serial connection and data is transmitted using USB port. The rosserial protocol can convert the standard ROS messages to embedded device equivalent data types, it also implements multi-topic support.

A package which enables us to run a ROS node at the embedded devices such as Nucleo is called the rosserial_client. On the PC side, we need some other packages to decode the serial message and convert to exact topics from the rosserial_client libraries. The recommended PC side node for handling serial data from a device is rosserial_python.

For high-level layer is realized by mini PC located on the robot. It is connected to Nucleo, laser scaner, IMU and GPS to receive sensor values which are essential for map building process using SLAM, obstacle avoidance and also the odometry as mandatory input to the navigational stack. We use standard ros messages inside i.e. sensor_msgs/LaserScan, sensor_msgs/Range, nav_msgs/Odometry.

We have applied the same Navigation Stack (as in Husarz robot) and used teb_local_planner node. It supports car-like kinematics, minimizes the trajectory execution time (time-optimal objective), provides separation from obstacles and compliance with kinodynamic constraints (i.e. maximum velocities and accelerations).

The base control system of Quadron robot is very similar to Husarz, and is shown in Figure 11. Most of the components are the same, the main differences are described below.

We have added a node called base_controller which communicates directly with the electronics of the robot. This node finally generates the motor speed

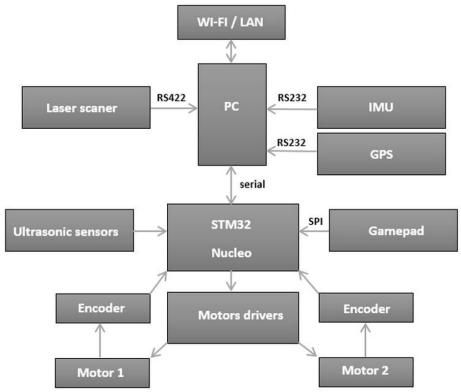

FIGURE 10. Controller of the Quadron robot

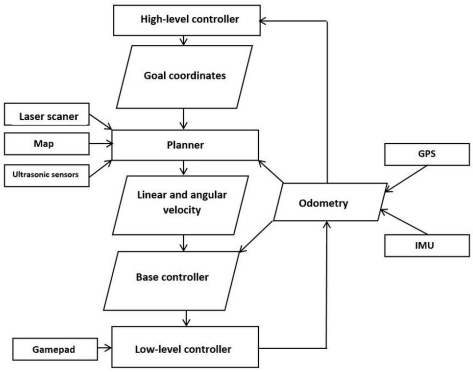

FIGURE 11. Navigation structure of the Quadron robot

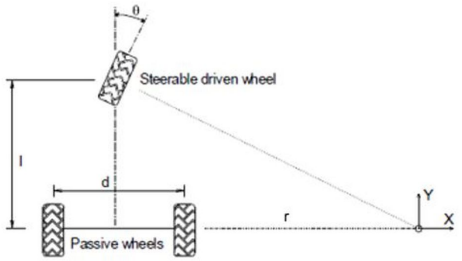

FIGURE 12. Kinematics of the Quadron robot

commands for each motor (PWM). Base_controller node subscribes to a topic cmd_vel, which is being published by the local planner and it converts standard message type geometry_msgs/Twist (which publishes linear v and angular Omega velocities) to the other type called ackermann_msgs/AckermannDrive according to the kinematic structure of the robot (see Figure 12) and using classic equations for Ackerman steering robots.

The base_controller node incudes also an implementation of PID controller, that is why it subscribes odomerty.

What is more, in the Quadron robot we used additional odometry sources i.e. IMU and GPS to obtain precise localization, and ultrasonic sensors in order to improve obstacle avoidance.

In case of any problems with high-level controller we are still able to control robot manually through gamepad which is connected directly to Nucleo board over SPI interface.

3. CONCLUSIONS

The great advantage of ROS is that we can supervise and visualize our data in real time. However, plotting is available only on x86 or x64 architecture because ARM processors (Raspberry Pi 3) do not support ROS graphic libraries. In our system the external computer is just a client that has an access to topics and can run its own nodes. The only requirement is to share the same network and configure the /.bashrc file to define the master's and client's IP addresses. Basically, it is the vital that we use appropriate forms of messages — specific message types and topic names to meet the nodes requirements. It is possible to remap topic names in launch files but it is somehow against ROS's philosophy that assumes focusing on preestablished design rules. This design concept makes ROS systems extremely extendable. Easy and logical file structure gives the opportunity to modify source code and add new nodes quickly. The growing ROS society is still supplying the repositories with a new software that can be used immediately after committing it to GitHub.

ACKNOWLEDGEMENTS

Research and participation in robotic contests supported by the Polish Ministry of Science and Higher Education under grant No. MNiSW/2016/DIR/199/NN

REFERENCES

[1] An official webpage of skaner. `http://skaner.p.lodz.pl`.

[2] I. Zubrycki, G. Granosik. Introducing modern robotics with ros and arduino, including case studies. *Journal of Automation, Mobile Robotics & Intelligent Systems JAMRIS* **8**(1):69–75, 2014. DOI:10.14313/JAMRIS_1-2014/9.

[3] P. Goebel. *ROS By Example*. Lulu, 2013.

[4] A. Martinez, E. Fernaandez. *Learning ROS for Robotics Programming*. Packt Publishing, 2013.

[5] Ros answers forum. `http://answers.ros.org`.

[6] Ros developers guide. `http://wiki.ros.org/DevelopersGuide`.

[7] Setup and configuration of the navigation stack on a robot. `http://wiki.ros.org/navigation/Tutorials/RobotSetup`.