TECHNISCHE UNIVERSITÄT WIEN	**Autonomous Racing Cars** **191.119 (VU 4,0) Semester: 2022S** **ROS Basics & Safety Node** **2021-03-15**

Student data

Matrikelnummer	Firstname	Lastname
12134689	Danilo	Castiglia
12135191	Artur	Chaves
01613004	Severin	Jäger
01552500	Johannes	Windischbauer

Transparency of contribution

- **Danilo Castiglia**: Report on ROS Messages, Python node creation and implementation
- **Artur Chaves**: Data Visualization, Report on data visualization and parameter adjustments
- **Severin Jäger**: Report on ROS workspace and package, package creation (incl. launch file) and debugging for C++ node, dynamic reconfiguration
- **Johannes Windischbauer**: Report on ROS Bags, C++ node pub/sub, calculation

1 ROS Basics

1.1 ROS Workspace and Package

a.) CMake[1] is a platform-independent build system. It uses `CMakeLists.txt` configuration files to generate build files. The `CMakeLists.txt` files contain required meta-information like the desired language version, dependencies, build options. This is then (in the case of C/C++ development on Linux systems) used to generate the corresponding makefiles automatically. Thus, `CMakeLists.txt` files contain more abstract information than makefiles as CMake takes care of platform-dependent subtleties.

b.) For ROS nodes written in Python, a `CMakeLists.txt` file is required as well. It handles messages, dependencies to other packages and tests. However, no executable file is created as CMake is only used to generate further Python code. Thus, the main Python file has to be made executable in the file system.

c.) `catkin_make` is always called in a catkin workspace which is usually located in `~/catkin_ws`.

d.) The `setup.bash` files set the environment variables required by the ROS tools to operate correctly including file paths and tool versions. The setup files `/opt/ros/noetic/setup.sh` an `devel/setup.sh` are identical besides the path of the Python script they call. The Python scripts are responsible for managing the environment variables and only differ in the `CMAKE_PREFIX_PATH` which in the former case only contains `/opt/ros/noetic` and in the latter case also `~/catkin_ws/devel`.

[1] https://cmake.org/

1.2 ROS Messages

a.)
```
/brake
/brake_bool
/clicked_point
/collision
/diagnostics
/drive
/dynamic_viz
/dynamic_viz_array
/env_viz
/env_viz_array
/gt_pose
/imu
/initialpose
/joy
/joy/set_feedback
/key
/map
/map_metadata
/map_updates
/move_base_simple/goal
/mux
/nav
/odom
/path_lines
/path_lines_array
/pose
/racecar/joint_states
/racecar_sim/feedback
/racecar_sim/update
/racecar_sim/update_full
/rand_drive
/rosout
/rosout_agg
/scan
/smoothed_path
/static_viz
/static_viz_array
/tf
/tf_static
/tree_lines
/tree_lines_array
/tree_nodes
/tree_nodes_array
/waypoint_vis
/waypoint_vis_array
```

b.) The output of the command is:

```
Type: sensor_msgs/LaserScan

Publishers:
 * /f1tenth_simulator (http://danilo-VirtualBox:44677/)
```

```
Subscribers:
 * /behavior_controller (http://danilo-VirtualBox:33659/)
 * /rviz (http://danilo-VirtualBox:36809/)
```

This command is used to retrieve informations about one topic. It gives us:

- the message type of the selected topic;
- the list of nodes that are publishing messages in the topic
- the list of nodes that have subscribed to receive messages from the topic

c.) Print messages continuously: `rostopic echo /scan`
Print just one message: `rostopic echo -n 1 /scan`[2]

d.) Looking at the message, the `angle_min` is -2.355 rad, the `angle_max` is 2.355 rad. The total angle (max - min) is 4.710 rad = 270 degrees, as stated in the previous assignment.
The message has a `range_min` of 0m and a `range_max` of 100m but we know from the previous assignment that the minimum distance of the Lidar is 0.06m and the maximum distance in ideal condition is 30m and in normal conditions is 10m, so only values in this range are significant.
Finally, the ranges array has 1080 values, that are all rays provided by the sensor (1081) except the ray at 0°.

e.) The message type for the topic `/scan` is `sensor_msgs/LaserScan`.
The structure[3] of this type is:

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

1.3 ROS Bags

a.) The command `rosbag record TOPIC1 [TOPIC 2 ...]` stores the `.bag` file in the directory it is called. To change the output directory and name you can either user the option `--output-prefix=PREFIX` or `--output-name=NAME`. If a rosbag is not named, the bag will get the current date-time as name. The prefix option prefixes the timestamp (which is often preferable), while the name option removes the timestamp altogether.

b.) Record /scan and /odom command: `rosbag record /scan /odom`.

[2] https://wiki.ros.org/rostopic
[3] http://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/LaserScan.html

2 Automatic Emergency Brake (AEB) with Time to Collision (TTC)

2.1 ROS Node: safety_node

The C++ source code for this task including the extensions for visualisation and reconfiguration is given in Listing 1 and the corresponding launch file in Listing 2. The Python node and the launch file are given in Listings 3 and 4 respectively. Two thresholds are used to determine whether to brake or not, one for moving forward, another (larger) one for going backwards.

2.2 Data visualization and parameter adjustments

a.) To visualise the data with `rqt_plot`, additional publishers (`pub_velocity_`, `pub_ttc_`, `pub_min_distance_`, `pub_brake_int_` in Listing 1) are required. The latter is necessary as `rqt_plot` does not support messages of type `Boolean`. Additionally, the corresponding messages have to be published regularly. An exemplary plot when the car encounters a wall is shown in Figure 1.

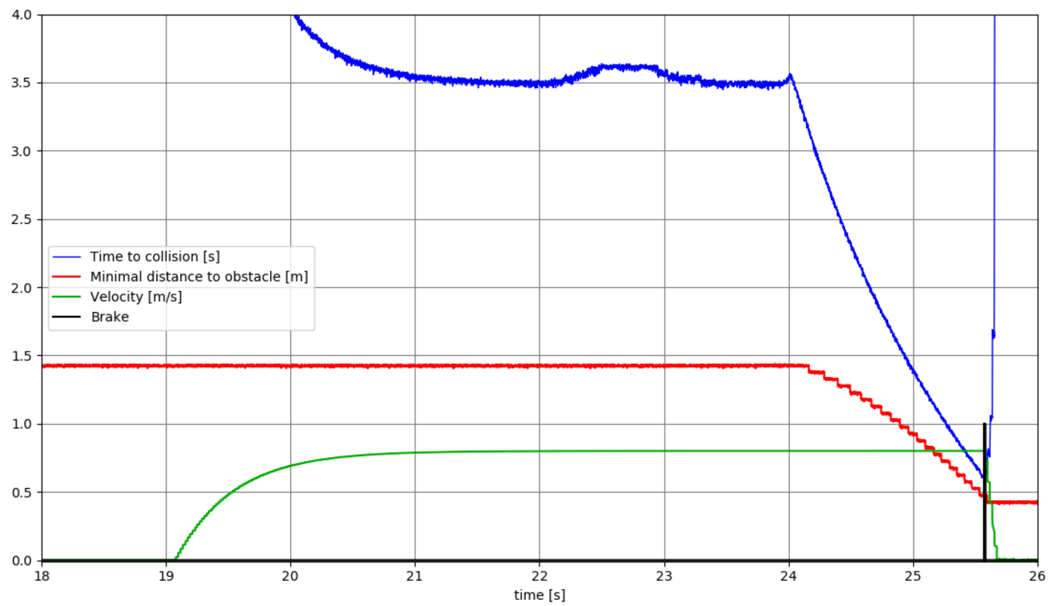
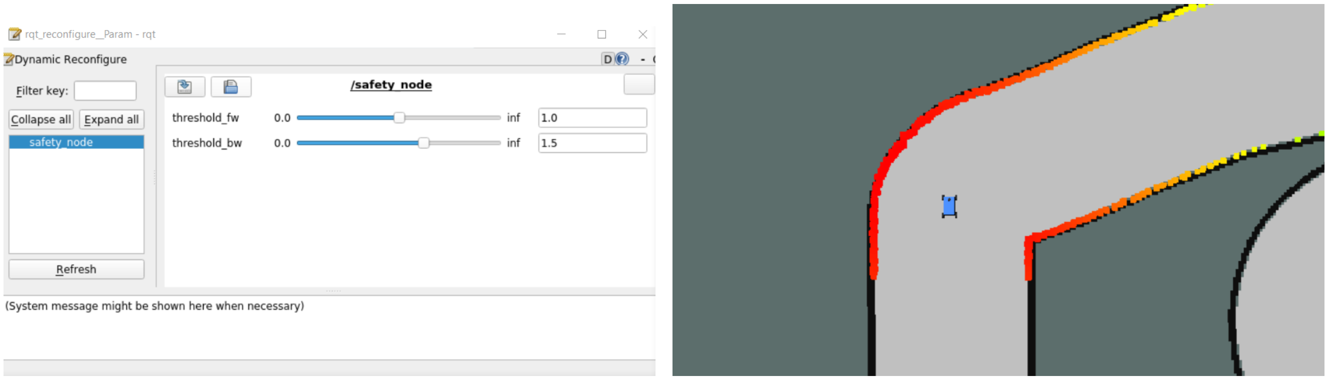


Figure 1: Data visualised with rqt plot

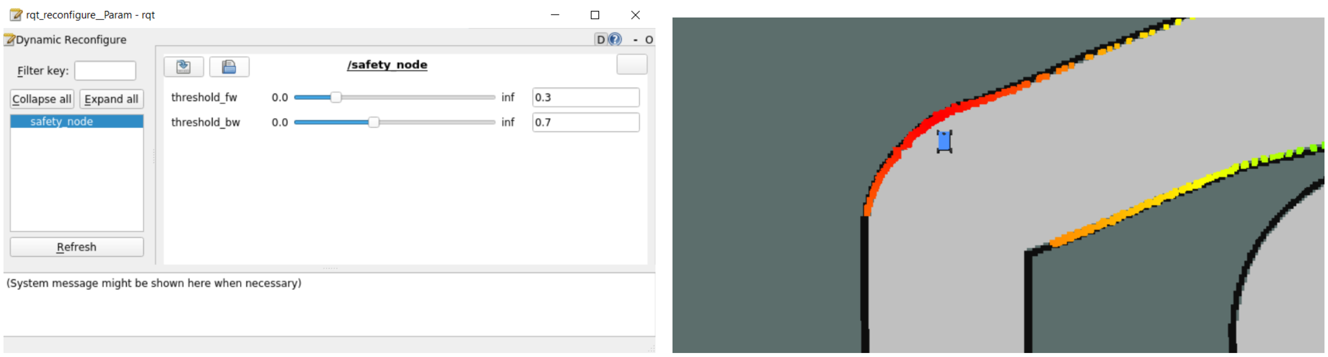
b.) Dynamic parameter reconfiguration requires a reconfiguration script (Listing 5) which defines the parameters including their names, types, and default values. After adding a `generate_dynamic_reconfigure_options` statement to the `CMakeLists.txt` file, this is used by the ROS toolchain to generate some code automatically. Then, both the reconfiguration server and the configuration defined in the script can be included, a reconfiguration server can be instantiated and a callback can be added (s. code highlighted with `Task 2.2.b` in Listing 1). Eventually, the callback adjusts the thresholds to the passed parameters whenever a change occurs.

Some results of adjusting the thresholds are visualised in Figures 2 and 3. A large threshold leads to safe driving, but induces undesired stops when driving parallel to a wall. Values around 0.6 for driving forward and 1.5 in the reverse direction are sufficiently safe and minimise false positives. However, in many situation even lower thresholds are feasible (like in Figure 3).



(a) rqt configure screen (b) Distance after emergency brake

Figure 2: C++ node with a forward threshold of 1



(a) rqt configure screen (b) Distance after emergency brake

Figure 3: C++ node with a forward threshold of 0.3

3 Appendix: Source Code

Listing 1: C++ implementation

```
#include <ros/ros.h>
#include <nav_msgs/Odometry.h>
#include <sensor_msgs/LaserScan.h>

// Include ROS msg type headers and libraries
#include <std_msgs/Bool.h>
#include <ackermann_msgs/AckermannDriveStamped.h>
#include <algorithm>
#include <math.h>

// Task 2.2.a
#include <std_msgs/Float64.h>
#include <std_msgs/Int32.h>

// Task 2.2.b
#include <dynamic_reconfigure/server.h>
#include <safety_node1/ThresholdsConfig.h>

double threshold_fw = 0.6;
double threshold_bw = 1.5;

// Task 2.2.b
void reconfig_callback(safety_node1::ThresholdsConfig &config, uint32_t level)
    ↪ {
    ROS_INFO("Reconfiguring␣TTC␣thesholds:␣fw:␣%f,␣bw:␣%f", config.threshold_fw
        ↪ , config.threshold_bw);
    threshold_fw = config.threshold_fw;
    threshold_bw = config.threshold_bw;
}

class Safety {
// The class that handles emergency braking
private:
    ros::NodeHandle n;
    double speed;
    double threshold;

    // Create ROS subscribers and publishers
    ros::Subscriber sub_scan_;
    ros::Subscriber sub_odom_;
    ros::Publisher pub_brake_;
    ros::Publisher pub_brake_bool_;

    // Task 2.2.a
    ros::Publisher pub_velocity_;
    ros::Publisher pub_ttc_;
    ros::Publisher pub_min_distance_;
    ros::Publisher pub_brake_int_;

public:
```

```
Safety() {
    ROS_INFO("Hello␣from␣safety_node!");

    n = ros::NodeHandle();
    speed = 0.0;
    /*
    One publisher should publish to the /brkake topic with an
    ackermann_msgs/AckermannDriveStamped brake message.

    One publisher should publish to the /brake_bool topic with a
    std_msgs/Bool message.

    You should also subscribe to the /scan topic to get the
    sensor_msgs/LaserScan messages and the /odom topic to get
    the nav_msgs/Odometry messages

    The subscribers should use the provided odom_callback and
    scan_callback as callback methods

    NOTE that the x component of the linear velocity in odom is the speed
    */

    // Create ROS subscribers and publishers
    sub_scan_ = n.subscribe("scan", 10, &Safety::scan_callback, this);
    sub_odom_ = n.subscribe("odom", 10, &Safety::odom_callback, this);

    pub_brake_ = n.advertise<ackermann_msgs::AckermannDriveStamped>("brake"
        ↪ , 10);
    pub_brake_bool_ = n.advertise<std_msgs::Bool>("brake_bool", 10);

    // Task 2.2.a
    pub_velocity_ = n.advertise<std_msgs::Float64>("velocity", 10);
    pub_ttc_ = n.advertise<std_msgs::Float64>("ttc", 10);
    pub_min_distance_ = n.advertise<std_msgs::Float64>("min_distance", 10);
    pub_brake_int_ = n.advertise<std_msgs::Int32>("brake_int", 10);
}

void odom_callback(const nav_msgs::Odometry::ConstPtr &odom_msg) {
    // Update current speed and TTC threshold
    speed = odom_msg->twist.twist.linear.x;
    if(speed < 0.0) {
        threshold = threshold_bw;
    } else {
        threshold = threshold_fw;
    }

    // Task 2.2.a
    std_msgs::Float64 velocity_msg;
    velocity_msg.data = speed;
    pub_velocity_.publish(velocity_msg);
}

void scan_callback(const sensor_msgs::LaserScan::ConstPtr &scan_msg) {
    int length = scan_msg->ranges.size();
```

```
double* ttc = new double[length];

// Task 2.2.a
std_msgs::Float64 min_distance_msg;
min_distance_msg.data = *std::min_element(&(scan_msg->ranges[0]), &(
    ↪ scan_msg->ranges[length-1]));
pub_min_distance_.publish(min_distance_msg);

// Calculate TTC
for (int i = 0; i < length; i++) {
    if (scan_msg->ranges[i] < scan_msg->range_min || scan_msg->ranges[i
        ↪ ] > scan_msg->range_max) {
        ttc[i] = std::numeric_limits<double>::infinity();
        continue;
    }

    double angle = scan_msg->angle_min + scan_msg->angle_increment * i;
    double r_derivative = -speed * cos(angle);
    ttc[i] = scan_msg->ranges[i] / std::max(-r_derivative, 0.0);

    // ROS_DEBUG("range %f at angle %f at speed %f yields ttc %f",
        ↪ scan_msg->ranges[i], angle, r_derivative, ttc[i]);
}

double min = *std::min_element(ttc, ttc + length);
ROS_DEBUG("min␣ttc=%f", min);

// Task 2.2.a
std_msgs::Float64 ttc_msg;
ttc_msg.data = min;
pub_ttc_.publish(ttc_msg);

// Publish drive/brake message
std_msgs::Bool brake_bool_msg;

// Task 2.2.a
std_msgs::Int32 brake_int_msg;
if (min < threshold && min > 0) {
    ROS_DEBUG("threshold␣warning␣with␣ttc=%f", min);

    ackermann_msgs::AckermannDriveStamped brake_msg;
    brake_msg.drive.speed = 0.0;
    pub_brake_.publish(brake_msg);

    brake_bool_msg.data = true;

    // Task 2.2.a
    brake_int_msg.data = 1;
} else {
    brake_bool_msg.data = false;

    // Task 2.2.a
    brake_int_msg.data = 0;
}
```

```
        pub_brake_bool_.publish(brake_bool_msg);

        // Task 2.2.a
        pub_brake_int_.publish(brake_int_msg);
    }
};

int main(int argc, char ** argv) {
    ros::init(argc, argv, "safety_node");
    Safety sn;

    // Task 2.2.b
    dynamic_reconfigure::Server<safety_node1::ThresholdsConfig> server;
    dynamic_reconfigure::Server<safety_node1::ThresholdsConfig>::CallbackType f
        ↪ ;
    f = boost::bind(&reconfig_callback, _1, _2);
    server.setCallback(f);

    ros::spin();
    return 0;
}
```

Listing 2: C++ implementation launch file

```
<?xml version="1.0"?>

<launch>
    <include file="$(find f1tenth_simulator)/launch/simulator.launch" />
    <node name="safety_node" pkg="safety_node1" type="safety_node" output="
        ↪ screen" />
</launch>
```

Listing 3: Python implementation

```
#!/usr/bin/env python3

import rospy
import math
from ackermann_msgs.msg import AckermannDriveStamped, AckermannDrive
from std_msgs.msg import Bool
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import Odometry


class Safety(object):
    """
    The class that handles emergency braking.
    """

    FORWARD_THRESHOLD = 0.6
    BACKWARD_THRESHOLD = 1.5

    def __init__(self):
        self.speed = 0
        self.acker = rospy.Publisher('brake', AckermannDriveStamped, queue_size
            ↪ =10)
```

```
        self.bool = rospy.Publisher('brake_bool', Bool, queue_size=10)

        rospy.Subscriber("scan", LaserScan, self.scan_callback)
        rospy.Subscriber("odom", Odometry, self.odom_callback)

    def odom_callback(self, odom_msg):
        self.speed = odom_msg.twist.twist.linear.x

        # If the car is stopped, then publish False to deactivate the emergency
           ↪  breaking
        if -0.005 <= self.speed <= 0.005:
            # Boolean message
            boolean = Bool()
            boolean.data = False
            self.bool.publish(boolean)

    def scan_callback(self, scan_msg):
        angle = scan_msg.angle_min
        i = 0
        while angle <= scan_msg.angle_max:
            beam_position = scan_msg.ranges[i]
            beam_speed = self.speed * math.cos(angle)
            if beam_speed <= 0:
                TTC = float('inf')
            else:
                TTC = beam_position / (beam_speed)
            i = i + 1
            angle = angle + scan_msg.angle_increment

            if (self.speed > 0 and TTC < self.FORWARD_THRESHOLD) or (self.speed
               ↪  <= 0 and TTC < self.BACKWARD_THRESHOLD):
                rospy.loginfo("TTC:" + str(TTC) + "␣under␣the␣threshold.")
                rospy.loginfo("Activating␣emergency␣brake")

                # AckermannDriveStamped message
                ack_drive = AckermannDrive()
                ack_stamped = AckermannDriveStamped()
                ack_drive.steering_angle = 0
                ack_drive.steering_angle_velocity = 0
                ack_drive.speed = 0
                ack_drive.acceleration = 0
                ack_drive.jerk = 0
                ack_stamped.drive = ack_drive
                self.acker.publish(ack_stamped)

                # Boolean message
                boolean = Bool()
                boolean.data = True
                self.bool.publish(boolean)

                break


def main():
```

```
    rospy.init_node('safety_node')
    sn = Safety()
    rospy.spin()


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

Listing 4: Python implementation launch file

```
<?xml version="1.0"?>

<launch>
    <node name="safety_node" pkg="safety_node2" type="safety_node.py" output="
        ↪ screen" />
    <include file="$(find f1tenth_simulator)/launch/simulator.launch" />
</launch>
```

Listing 5: Dynamic reconfiguration script for the C++ node

```
#!/usr/bin/env python
PACKAGE = "safety_node1"

from dynamic_reconfigure.parameter_generator_catkin import *

gen = ParameterGenerator()

gen.add("threshold_fw", double_t, 0, "TTC␣threshold␣if␣driving␣forward",
   ↪ default=0.6,  min=0)
gen.add("threshold_bw", double_t, 0, "TTC␣threshold␣if␣driving␣backward",
   ↪ default=1.5,  min=0)

exit(gen.generate(PACKAGE, "safety_node", "Thresholds"))
```