Documentation of Lab work for group 8

Wiktor Kochanek
Electronic Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany
wiktor.kochanek@stud.hshl.de

Vytaras Juraska
Electronic Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany
vytaras.juraska@stud.hshl.de

Hermann Anguiga
Electronic Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany
hermann.anguiga@stud.hshl.de

Alexander Wilms
Electronic Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany
alexander.wilms@stud.hshl.de

***Abstract*—This document is a record of our group first forray into the world of autonomous driving.**

I. Introduction

Motivated by a class assignement our group was tasked with creating a self-driving vehicle. We achieved this by using line tracking sensors and ultrasonic sensors, but also attempted to implement a camera and video streaming as the next step.

II. Analyzing the specifications

Our assignment came with clearly defined specifications:

- Line tracking system
- Ultrasonic sensor integration
- Camera streaming system

We split these objectives into two sections: car focused - line tracking and ultrasonic sensors systems - and camera focused - the streaming. We then split the tasks inside our group so that each member can focus on a specific objective to bring the best possible results.

A. Car focused specifications

1) Line tracking system: The car was supposed to be able to follow a line around a track using two line tracking sensors. The specific implementation of such function was left to our own choice. We decided to make a system that would allow the car to drive when it doesnt detect a line and react when it does detect a line.

2) Ultrasonic sensors system: The car was supposed to integrate a system using three ultrasonic sensors - what said system would influence was left to us to decide. we settled on an obstacle avoiding system that would co-operate with the line tracking system - if there was an obstacle on the track the car would attempt to drive around the obstacle and drive back onto the track.

B. Camera focused specification

The car was supposed to integrate a camera and a system that would allow it to stream the camera view onto an external computer. for this task we used a Raspberri Pi with a camera module and a Mac computer.

III. Architecture

A. Line tracking sensors

The infrared (IR) sensor used for this project is mainly made of a transmitting IR-LED, a receiving photodiode, an adjustable resistor, a hole for fixation and connecting pins. This IR sensor works by detecting reflected light from its own LED. By measuring the amount of reflected infrared light, it can detect transitions from bright to dark objects. When the sensor

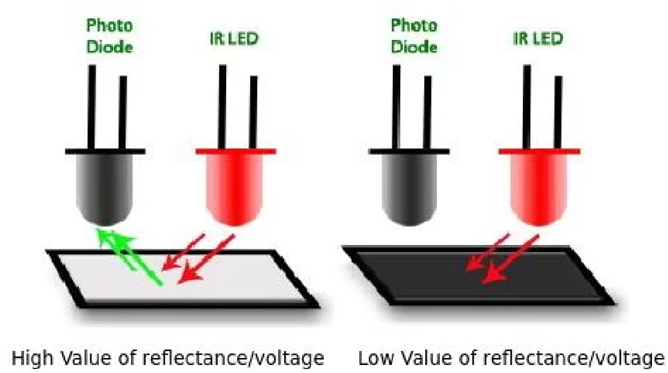

Fig. 1. Demonstration of how line tracking sensor works

receives an appropriate voltage, the IR-LED emits infrared light. That light travels from the car to the surface on which it stands. If the surface is dark, the emitted light is absorbed, then there is no reflection. The photodiode diode senses nothing. There is no signal out of the sensor. If the surface is transparent the situation stays the same as previously mentioned except here light passes through the surface. Finally, if the surface is reflective, the emitted light bounces back and is detected by the photodiode. A signal is generated and transmitted to another component of the circuit, in the present case: a microcontroller. We would like to emphasize that to create a sensible detection, IR LED should be used with a low value resistor so that it shines brighter. Also, the color and reflective proprieties of the surface play significant roles on the quality of the detection.

B. Ultrasonic sensors

To avoid crashing into a structure, the car is equipped with an obstacle detection system made of three ultrasonic sensors that are positioned at the front part of the car. One sensor is located at the center and the other two on the sides.

For this project, we made use of a particular ultrasonic sensor named HC-SR04. The HC-SR04 has two main components: A transmitter and a receiver. It has also four pins for connections. Namely VCC, Trig Echo and GND.

During its operation, the HC-SR04 is most accurate when the object to be detected is directly in front of it. Object within 15 degree left and right get also a reasonable detection. Outside this 30 degree range, it doesn't function.

The short version of the operating mode of The HC-SR04 is as followed: When a certain voltage is applied to the trigger pin, the sensor transmits a burst of eight pulses. While these pulses are travelling in airway, the Echo pin goes high and forms an echo-back signal. If the transmitted pulse is not reflected back, within a certain amount of time, the echo-back signal timeout and fades out. An internal signal is then produced to indicate there is no obstruction within the range of the sensor. But if the pulse is reflected back, an internal signal is produced at the echo pin (Fig. 2.).

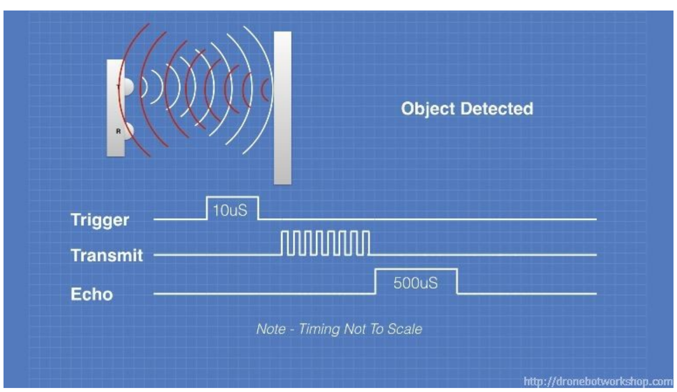

Fig. 2. Demonstration of how the ultrasonic sensor works

This sensor cannot actually tell distance - it can only measure the time it took for the pulses to come back to the sensor - as such, the calculation of distance needs to be done in the code. We take the time the sensor gives is in microseconds and convert it to distance using this formula:

$$s = t * 0.034/2 \tag{1}$$

Where t is the time for the wave to travel and s is the calculated distance. The time is divided by 2 as the wave needs to travel the distance twice - out and then back.

C. Motors

A DC motor is a machine that transforms electrical energy into mechanical energy in form of rotation. Its physical outside appearance consists of a shaft a body and connecting terminals. When a voltage is applied to a motor it runs. If the polarity is inverted, the motor changes its direction. Finally, if the motor is running and both terminals are disconnected, it will keep rotating but will certainly slow down and stop. Knowing these conditions has eventually dictated our thinking on how to control this car. From the code point of view motors are very simple to use. We attach a PWM value to them using a digitalWrite() function to determine the % of power they should run at.

D. RaspberryPi

For this project in the case of the video streaming component, we used the versatile single-board-computer named RaspberryP Model 3. The RaspberryPi series in general is worldwide used and shows the practicality in many different use cases globally. One of the main points is that it is promoted to teach the basics of computer sciences in schools, universities and in developing countires. Most of our group members have heard or even worked with a RaspberryPi before, which is a plus. However it is simple to understand and the initial training is fast. As it is a linux based system the commands and functions are simliar to other linux systems. The flexibility of this mirocomputer makes it perfect for rapid protyping in genreral but also perfect for our creation of a self-driving vehicle because we are able to adapt and change parts in the future. The RaspberryPi is also used in the domain of informatics, robotics and electronic engineering but is also interesting for hobbyists who want build projects for themselves. The small price point of under 40 Euros makes it suitable for a lot of people getting in the field. This is a motivation for us to use it and see what the hardware is capable of with our knowledge and work. This way we can publish our code to GitHub and make it public, so other people and students can build their version of a self driving vehicle with a camera stream. It is commonly kown that many people are using and have used the RaspberryPi for surveilance systems, video recordings and streaming projects. We want to give it a try and see if the project is usable for car streaming scenario with a short delay. As stated before the car streaming is realized on a RaspberryPi 3. The hardware parts, input/output possiblities and specs are quite remarbale for its size and price. The project definitely benefits from these parts but it has to be noted that the RPi Model 3 might be overpowered for our simple stream of 320x240 pixels. Nevertheless the power of the hardware could be useful for future adaptations. The RaspberryPi features a 64-bit Broadcast System on a chip (SoC) with an integrated ARM-compatible central processing unit (CPU) and on chip graphic processing unit (GPU). The Processor Speed is 1.4 GHz and an on-board memory of 512 MB SDRAM. It also features a Bluetooth and Wifi functionality used for the Transport Control Protocol (TCP) stream. The Rpi has a 5V/2.5A DC power input which is connected to a power bank.

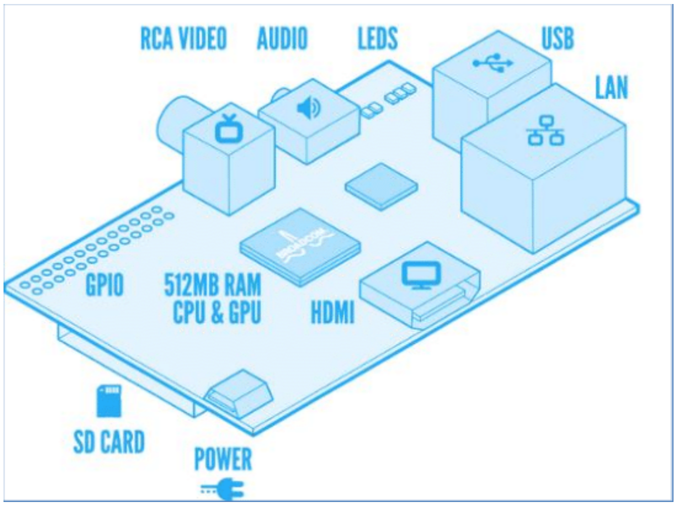

Fig. 3. Rough Architecture of RPi

The Extended 40-pin GPIO header was not used but could be useful in further modifactions. The streaming aspect was realized with the CSI camera port and the Camera Module v2. The Full Size HDMI Port can be to connect to a PC monitor to change parameters or the code. It could also be used to stream videos/images directly on a smaller display mounted on the car.

E. Camera Module

The focus of a autonomous car/selfdriving vehicle is to process information and react fast. To get to this point, the engineering aspect requires optimized hardware and software components. We used the powerful Microcomputer RaspberryPi and the official RaspberryPi camera module v2. These

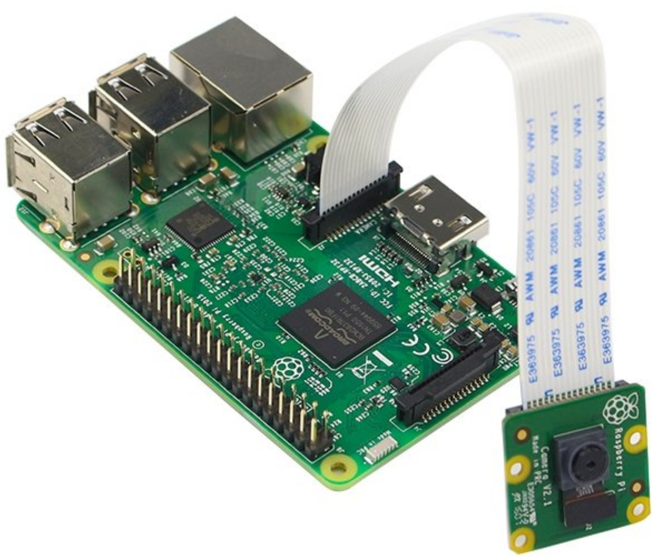

Fig. 4. A Connected Camera Module to RPi

components are connected via ribbon cable to the special CSI camera port for optimized compatibility. The camera in it self is small and well constructed. It has a Sony IMX219 8-megapixel sensor. The Module v2 can be used to used to take still images as well as high-defintion video. It supports 1080p 30fps, 720p 60fps and VGA 90 fps video modes. The camera is easy to use but has many functions for more complicated tasks. There are lots of examples and tutorials on GitHub and other websites, where people show of the capability of the camera by using it for time-lapse, slow-motion and other video effects. There is also the possibility to use libraries and open source projects.
The car requires a fast and reliable stream for the user and system. This is why we decided to use lightweight protocols like TCP, Berkley Sockets and a small but suffiecient image size of 320x240 pixels. This way we reached a delay of less than 50ms. But before the camera system can be used, setting up the device has to be done. The RaspberryPi has many suitable operating systems, but we chose the official distribution ”Raspbian” because it is the most supported and the most used operating system for RPi. After downloading the ”Raspian Stretch with Desktop” operating system in a format of a .zip file, the image needs to be written on the SD card. We decided to use the program called ”Balena Etcher” which, after some complications, in the end worked as intended. After the image is written, and a functional and bootable operating system has been successfuly launched, it is required to enable the camera software, in order to use the camera module. This can be done by the following steps: in the ”Prefernces menu” there is a ”RaspberryPi Configuration Window”, in the ”Interface Tab” is the option to ”enable Camera”. After these steps are completed and reboot is done, the RPi is ready to recognise, use and grant access to the Camera Module.

IV. System Design

A. Car focused System Design

The car will operate within 4 states for line tracking system and a pseudostate for the ultrasonic sensors sytem. Different enviromental inputs will influence the behaviour of the car. The car will always begin in the line tracking state attempting to go forward. When one of the line tracking sensors detects that it has gone onto the line it will steer the car to get the sensor back off the line - Turn Right and Turn Left states. If it happens that both sensors are on the line at the same time the car will stop. Should the cars ultrasonic sensors detect an obstacle it will enter into the Avoiding Obstacles Pseudostate. It will then decide if the obstacle is closer to its right or left side and will attempt to avoid the obstacle on the other side. When it succesfully returns to the track, the line tracking sensors will exit the car from the pseudostate and will reutrn line tracking functionality.

B. Video steam focused System Design

Since specifications did not define what methods of implementation had to be used, we really wanted to implement our video streaming system on to OpenCV, hence our system design was focused more on the implementation methods with

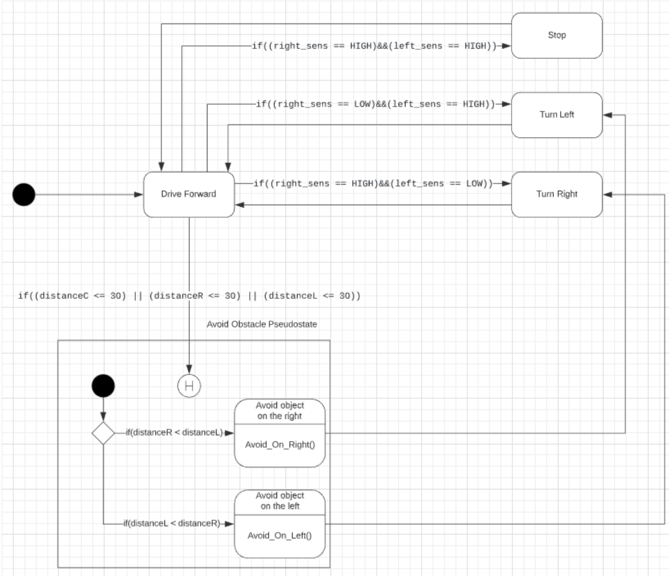

Fig. 5. UML State Diagram

OpenCV. After some reaserch of how does the main concept of video streaming look like, we decided to settle on our finalised base concept of our video streaming system design. The idea

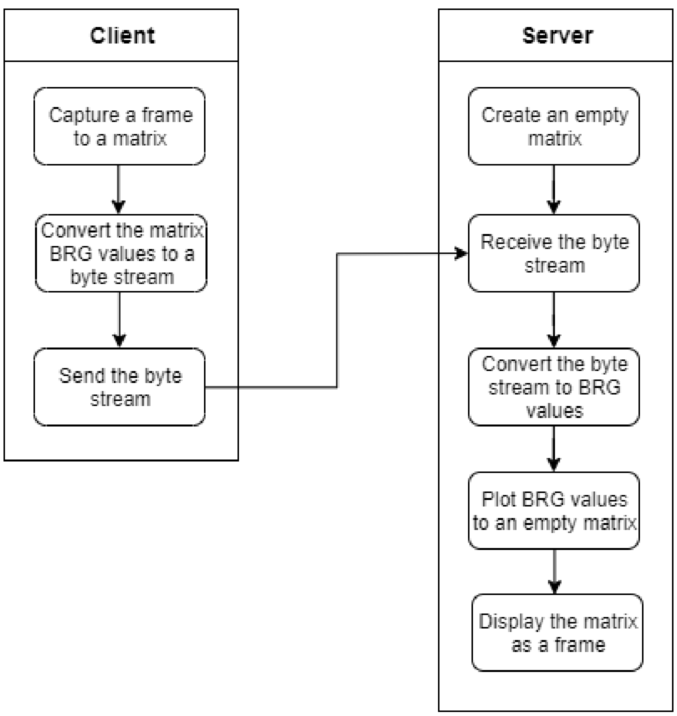

Fig. 6. Video Streaming System Design

for the client was to portray frames as matrices, in which every single pixel would be a component with 3 channels in the sequence of Blue, Red and Green. We portray each channel in a byte value with a limited amount of bits dedicated to each channel component. So our client has to store each frame in a matrix, convert every single matrix component channel into a byte stream and send the byte stream over to the server. From the server side the process is a bit lengthier, first it has to create an empty matrix, as a template. After the server receives the byte stream from the client, it has to convert the byte stream back to the channels, which would resemble each component in the matrix. After plotting each component sequentially back to the matrix template, we would receive a whole frame, which we finally can open up as an image.

V. Implementation of the system

A. Line tracking system

The line tracking system was the first thing tackled in the car functionality. The first implementation was a simple use of ”if” and ”if else” functions - depending on which line tracking sensor was tirggered, speeds of motors would change to make the car drive forward, turn the correct way or stop completely. Later implementation made use of interrupts, which changed the motor speed in the interrupt service routine. [1] While the

```
void left_turn(){
   analogWrite(motorPin2, 255);
   analogWrite(motorPin4, 64);
   }
```

Fig. 7. used interrupt service routine[1]

interrupt solution was a great way to make the source code easy to understand and the car ran very smoothly, it was discarded during integration with ultrasonic sensor code as there were compatiblity problems. As such we settled on a ”if” and ”if else” functions solution.

B. Obstacle avoidance system

Implementation of the ultrasonic sensors was not specifically defined in the project requriements. Our design used them in a way to detect and obstacle in the way and attempt to drive around it. First the car had to know that there was an obstacle in front of it. This was achieved simply using an ”if” function - if any of the three sensors detect an obstacle within 30cm of the car they will trigger the if condition and enter the car in the pseudostate of ”avoid obstacle”. After the car enter

```
else if((distanceC <= 30) || (distanceR <= 30) || (distanceL <= 30)){
```

Fig. 8. if condition detecting the obstacle

this pseudostate we have to choose which way is best to avoid the obstacle on - meaning that if theres a wall on our right, we want to turn to the left to clear it and vice versa. This is simply done by comparing the distance values on the right and left ultrasonic sensors - distanceR and distanceL respectively. In

[1]lines of code in Fig. 2 do not represent actual content in the final source code

this case we do not care about the centre distance as it doesnt influence our choice of side to avoid on. After the choice is made the car enters a loop that lets it go around an obstacle - in this example we will be avoiding an obstacle on the right (distanceL < distanceR). After starting the avoiding algorithm

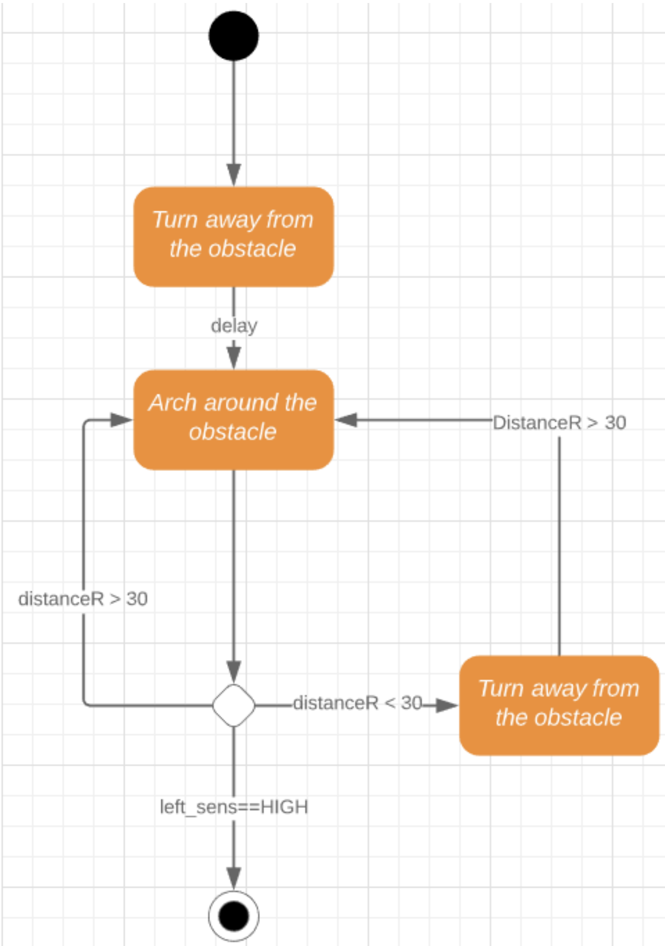

Fig. 9. Activity diagram of the avoiding algorithm

the car will slowly arch around its obstacle. it is possible that the obstacle is longer than the arch diameter, so the car is able to detect obstacles in this mode and turn away again if needed. This algorithm was designed to allow the car to leave its line in order to avoid the obstacle and detect when it is supposed to once again follow said line. In order to do that we had to have the car ignore one of the line tracking sensors for the duration of avoiding algorithm - in this case the right line tracking sensor - and then respect the sensor again when it gets back on the line. This was simply done by running the algorithm in a subfunction and including a simple if condition in the loop. The car arching around the obstacle is implemented using a while-loop that relies on a variable set after the car decides on which side is the obstacle. The only way to stop that loop is the active line tracking sensor - in this case the left one - to cross onto the line. The sensor triggering ends the loop and the avoidance algorithm returning the car to its line tracking functionality.

C. *Video stream system*

Since we wanted to implement OpenCV into our video streaming system, we decided to make it in C++ programing language. As a method of communication, with no doubts we chose TCP Internet Protocol and its implementation in C++ programming - Berkeley Sockets, which we, at the time, recently learned from the lectures and were really motivated to implement it into our video streaming.

The first problem we encountered, was what to use, to compile OpenCV and Berkeleys Sockets in C++. After trying out many popular IDE's (Integrated Development Enviroment) on a Windows based device, we decided to start the development purely on our Raspberry Pi, which was running on Raspbian, Debian-based opperating system. Since we finally started working on Raspbian, all we had to do, was to install OpenCV libraries, dedicated for Raspberry Pi, and compiling was not an issue anymore. After we could finally compile our code properly, we started learning how Berkeley Sockets worked and how to implement a client and a server successfuly, since we had implementation in our lectures, the process was not complicated, but fully understanding it and changing properties in a way, that the communication would function as intended was challenging. Once we understood the basics of Berkeley Sockets, we started implementing OpenCV to Berkeley Sockets. Since we had a bit of prior experience with OpenCV, we did not have to learn it from the basics, so we instantly went for implementation. After countless trial and error, we finally got it to work:

1) Client system: Before any task was performed, the client has to check if the camera was aviable. That was easily done by implementing OpenCV class, called "VideoCapture" which

```
VideoCapture cap;
if(!cap.open(0)) {
      printf("Camera doesn`t work");
      return 0;
}
```

Fig. 10. Camera access check

has many functions dedicated for camera access. One of the functions is called "open", which if parameters of the funcion "open" are not zero, that means there was no access to the camera or there was an error.

Going forward the code enters an infinite loop, which would be the whole process of connecting, capturing frames, converting to byte stream, sending byte stream. First we begin from creating a Socket. We define what a socket is and all of its properties are, simply put, referring to the type of protocol, which has to be used, which in our case is TCP. Later on we check, if we were successful in creating a socket.

Afterwards, using "inet_pton" function, we have to connect to the server knowing the socket address, port and its IP (Internet

```
SocketFD = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
if (SocketFD == -1) {
  perror("Failed to create a socket");
  exit(EXIT_FAILURE);
}
```

Fig. 11. Socket creation in Client

Protocol). Just then we can finally connect our socket using the ”connect” function.

```
res = inet_pton(AF_INET, IP, &sa.sin_addr);
connect(SocketFD, (struct sockaddr *)&sa, sizeof sa);
```

Fig. 12. Connecting to Server and connecting the Socket

And if we finally have connected to the server, we can start capturing the frame, converting it and sending it to the client. First we create a matrix, which we use to store the frame,

```
Mat image;
cap >> image;
resize(image, image, Size(320,240));
long imgSize = image.total()*image.elemSize();
send(SocketFD, reinterpret_cast<char*>(image.data), imgSize, 0);
```

Fig. 13. Capturing the frame, converting and sending to Server

when we capture it with already predefined ”VideoCapture” class variable. We resize the image to 320 by 240 pixels to improve the speed of sending the byte stream and calculate the final image size, which is later on needed in the ”send” function. The final function ”send” is a bit more tricky, first we dedicate the already connected socket as a socket for sending, dedicate the message property, which in our case are matrix components converted to a long byte stream, and shortened by ”reinterpret_cast”, define the already calculated size of the message, which is the size of the frame, and declare the flag, which was never required, so we left it at zero. The tricky part of ”send” function is the ”reinterpret_cast”, because this type of operator is not safe, it does not check if pointer type and the data, which the pointer is pointing at, are the same or not, which can harm the device, if an error in pointers occur, or the types of variables in the code are mixed up.
That is where a single cycle of the loop ends and this is the core of the client system.

2) *Server System:* Majority of functions, which are in the client system, are reused in the server system as well. Firstly (same as in Fig. 7), we create a TCP socket and check, if the creation was successful. Then we try to bind the socket to the socket address and the dedicated port, through which our client will try to connect to the server. If the binding has failed (for example the port is in use), then we close the socket, we've been trying to open, and terminate the program.
This is where the infinite loop from the server side starts and its starts of with the function ”listen”. Its properties, are

```
if (bind(SocketFD,(struct sockaddr *)&sa, sizeof sa) == -1) {
  perror("bind failed");
  close(SocketFD);
  exit(EXIT_FAILURE);
}
```

Fig. 14. Binding the socket

the socket, which to check for incoming connections, and the maximum incoming connections allowed at a single time period. If ”listen” function returns negative one, it means, that some sort of error has been made and the socket has to be closed, and the program has to be terminated for security reasons.

```
if (listen(SocketFD, 10) == -1) {
  perror("listen failed");
  close(SocketFD);
  exit(EXIT_FAILURE);
}
```

Fig. 15. Listening error check

Then we create the zero matrix, which is a template with components, which are all empty. We calculate an estimate size of the matrix, to have an exact expected maximum byte stream size, when the server receives it. Before communication both client and the server have to agree on the resolution of the frames, for the frame to be recollected properly from the byte stream. Then we store the recieved data to a variable with a function ”recv”, which properties are: socket from where it recieves the byte stream, the byte stream itself, the estimate size of the byte stream from the predetermined zero matrix.
One of the last steps of the loop is the byte stream conversion to the matrix components channel values. The byte stream is stored in a corresponding sequence of channels: Blue, Red, Green. We divide the whole byte stream into segments, where one segment represents one channel value, three of those channels represent a single component of the matrix, which in the end is just a pixel in a frame. So this exact part of the loop divides a byte stream to pixels and sorts them row by row, column by column back into a single frame.

```
int ptr=0;
for (int i = 0;  i < img.rows; i++) {
  for (int j = 0; j < img.cols; j++) {
    img.at<cv::Vec3b>(i,j) = cv::Vec3b(sockData[ptr+ 0],sockData[ptr+1],sockData[ptr+2]);
    ptr=ptr+3;
    }
}
```

Fig. 16. Byte stream conversion to BRG values

And the last part of the loop is opening up an already prepared frame with a function ”imshow”, after a single frame is shown, it has to close after some time, so we have to implement a function ”waitKey”, which waits for a specified time in miliseconds or completely terminates the program if any key on the keyboard was pressed. If ”waitKey” wouldnt be implemented, more and more frames would stack on each other, taking up all the recourses of the device, so this function is necessary.

In the end TCP solution proved to be remarkable in results. Using a Mac and the local Wi-Fi we achieved latencies as small as 20-50ms - very good results for a system designed for a drone-like car. TCP was also a good choice for its simplicity, small image sizes and the previously mentioned speed.

VI. Group Evaluation

In general we think this project was a great way to develop teamwork skills but also to learn about about the development processes. We used GitHub to apply versioning methods, which made the work a lot easier to coordinate and to quickly ship improvements to the systems. A major surprise of the project was how quickly we got the first version to run - the car was assembled in less then an hour and could drive shortly after. The work we have done in the labs prior to the project assignment definitely helped with that - getting to understand the Arduino and the components individually first was a major factor in us able to work at this rate The implementation of the line tracking system also was done very quickly and worked surprisingly well - it was a really good prototype project to understand the workings of such systems. Even though big portions of the program were outsourced from different implementations and slightly modified to fit our needs and requirements, the results software wise were astonishing. The longer we spent trying to figure out what were missing and what we already have in our code, the more offten we came back to the UML diagrams, the more we started to understand our requirements, what we need to do to achieve them, and how different parts work, when going more into depth of implementation and design. Unfortunately, majority of functions used were parts of different libraries, which we did not thrive to understand, because of time pressure. Never the less, we are still curious and will continue being interested in the topics, which this project covered. It feels like we are just barely starting to understand the possibility of developing and engineering. With that being said we believe that there was much room for improvement in all areas of the project and given more time - or if we managed it better - this could’ve been a much more advanced implementation.

VII. Annex

Work done in % by each member of the team:

- Wiktor - 25%
- Vytaras - 25%
- Hermann - 25%
- Alex - 25%

Work done in total hours by each member of the team:

- Wiktor - 40hrs
- Vytaras - 40hrs
- Hermann - 40hrs
- Alex - 40hrs

Below you can find the car system code:

```
//Line Sensors
const int right_sens = 2;
const int left_sens = 3;
int right_sens_override = 0;
int left_sens_override = 0;

//Motor 1 the right one
const int motorPin1 = 8; //backwards
const int motorPin2 = 9; //forwards

//Motor 2 the left one
const int motorPin3 = 7; //backwards
const int motorPin4 = 6; //forwards

//Ultrasonic Sensors
const int trigPinC = 13;
const int echoPinC = 12;
long durationC;
int distanceC;
const int trigPinR = 5;
const int echoPinR = 4;
long durationR;
int distanceR;
const int trigPinL = 11;
const int echoPinL = 10;
long durationL;
int distanceL;

void setup(){

  Serial.begin(9600);
  pinMode(right_sens, INPUT);
  pinMode(left_sens, INPUT);

  pinMode(motorPin1, OUTPUT);
  pinMode(motorPin2, OUTPUT);
  pinMode(motorPin3, OUTPUT);
  pinMode(motorPin4, OUTPUT);

  pinMode(trigPinC, OUTPUT);
  pinMode(echoPinC, INPUT);
  pinMode(trigPinR, OUTPUT);
  pinMode(echoPinR, INPUT);
  pinMode(trigPinL, OUTPUT);
  pinMode(echoPinL, INPUT);
}


void loop(){

  USC_Sensor();
  USR_Sensor();
  USL_Sensor();

  if((digitalRead(right_sens) ==
      LOW)&&(digitalRead(left_sens) ==
      LOW)){//forward

    analogWrite(motorPin2, 191);
    analogWrite(motorPin4, 191);
  }
```

```
  if((digitalRead(right_sens) ==
      LOW)&&(digitalRead(left_sens) == HIGH)){

   analogWrite(motorPin2, 255);
   analogWrite(motorPin4, 64);
  }

  if((digitalRead(right_sens) ==
      HIGH)&&(digitalRead(left_sens) == LOW)){

   analogWrite(motorPin2, 64);
   analogWrite(motorPin4, 255);
  }

  if((digitalRead(right_sens) ==
      HIGH)&&(digitalRead(left_sens) ==
      HIGH)){//stop

   analogWrite(motorPin2, 0);
   analogWrite(motorPin4, 0);
  }
  else if((distanceC <= 30) || (distanceR <=
      30) || (distanceL <= 30)){//obstacles

   analogWrite(motorPin2, 0);
   analogWrite(motorPin4, 0);
    if(distanceR < distanceL){
     right_sens_override = 1;
     Avoid_On_Right();
    }
    else if(distanceL < distanceR){
     left_sens_override = 1;
     Avoid_On_Left();
    }
  }

}

void Avoid_On_Right(){
   USR_Sensor();
   analogWrite(motorPin2, 127);
   delay(1500);
   analogWrite(motorPin2, 0);

while(right_sens_override > 0){
   USR_Sensor();
  if(distanceR < 50){
   analogWrite(motorPin2, 127);
   analogWrite(motorPin4, 0);
  }else{
   analogWrite(motorPin2, 64);
   analogWrite(motorPin4, 127);
  }if(digitalRead(left_sens) == HIGH){
   right_sens_override = 0;
  }
}
}

void Avoid_On_Left(){

  USL_Sensor();
  analogWrite(motorPin4, 127);
  delay(1500);
  analogWrite(motorPin4, 0);

while(left_sens_override > 0){
  USL_Sensor();
  if(distanceL < 50){
   analogWrite(motorPin4, 127);
   analogWrite(motorPin2, 0);
  }else{
   analogWrite(motorPin2, 127);
   analogWrite(motorPin4, 64);
  }
  if(digitalRead(right_sens) == HIGH){
   left_sens_override = 0;
  }
}
}

void USC_Sensor(){

  digitalWrite(trigPinC, LOW);
  delayMicroseconds(2);

  digitalWrite(trigPinC, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPinC, LOW);

  durationC = pulseIn(echoPinC, HIGH);

  distanceC = durationC*0.034/2;

  Serial.print("DistanceC: ");
  Serial.println(distanceC);
}

void USR_Sensor(){

  digitalWrite(trigPinR, LOW);
  delayMicroseconds(2);

  digitalWrite(trigPinR, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPinR, LOW);

  durationR = pulseIn(echoPinR, HIGH);

  distanceR = durationR*0.034/2;

  Serial.print("DistanceR: ");
  Serial.println(distanceR);

}

void USL_Sensor(){

  digitalWrite(trigPinL, LOW);
  delayMicroseconds(2);

  digitalWrite(trigPinL, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPinL, LOW);

  durationL = pulseIn(echoPinL, HIGH);

  distanceL = durationL*0.034/2;

  Serial.print("DistanceL: ");
  Serial.println(distanceL);

}
\end{lstlising}
Below you can find the camera stream code ran
```

on the server side:

```
\begin{lstlisting}
#include <sys/types.h>
  #include <sys/socket.h>
  #include <netinet/in.h>
  #include <arpa/inet.h>
  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include <unistd.h>
  #include <iostream>

  #include <opencv2/opencv.hpp>

  using namespace cv;
  using namespace std;

  int main(void)
  {
    Mat image_recieved;
    struct sockaddr_in sa;
    int SocketFD = socket(PF_INET,
        SOCK_STREAM, IPPROTO_TCP);
    if (SocketFD == -1) {
      perror("Can not create a Socket");
      exit(EXIT_FAILURE);
    }

    memset(&sa, 0, sizeof sa);

    sa.sin_family = AF_INET;
    sa.sin_port = htons(1100);
    sa.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(SocketFD,(struct sockaddr *)&sa,
        sizeof sa) == -1) {
      perror("bind failed");
      close(SocketFD);
      exit(EXIT_FAILURE);
    }


    for (;;) {

      if (listen(SocketFD, 10) == -1) {
        perror("listen failed");
        close(SocketFD);
        exit(EXIT_FAILURE);
      }
      int ConnectFD = accept(SocketFD, NULL,
          NULL);

      if (0 > ConnectFD) {
        perror("accept failed");
        close(SocketFD);
        exit(EXIT_FAILURE);
      }

      Mat img = Mat::zeros(320,240, CV_8UC3);
      long imgSize = img.total()*img.elemSize();
      char sockData[921600];
      long bytes;
      //Receive data here

      for (int i = 0; i < imgSize; i += bytes) {
        if ((bytes = recv(ConnectFD, sockData
            +i, imgSize - i, 0)) == -1) {
          }
        }

        // Assign pixel value to img
        int ptr=0;
        for (int i = 0; i < img.rows; i++) {
          for (int j = 0; j < img.cols; j++) {
            img.at<cv::Vec3b>(i,j) =
                cv::Vec3b(sockData[ptr+
                0],sockData[ptr+1],sockData[ptr+2]);
            ptr=ptr+3;
            }
        }


        resize(img, img, Size(640,480));
        namedWindow("Stream");
        imshow("Stream", img);
        waitKey(30);
      }
      destroyAllWindows();
      close(SocketFD);
      return EXIT_SUCCESS;
}
\end{lstlising}
```

Below you can find the camera stream code ran
    on the client side:

```
\begin{lstlisting}
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <string>
#include <iostream>

#include <opencv2/opencv.hpp>

using namespace std;
using namespace cv;
int main(void)
{
  const char *IP = "172.31.12.102";

  struct sockaddr_in sa;
  int res;
  int SocketFD;
  VideoCapture cap;
  if(!cap.open(0)) {
       printf("Camera doesn't work");
       return 0;
  }
  for (;;) {
       SocketFD = socket(PF_INET, SOCK_STREAM,
           IPPROTO_TCP);
       if (SocketFD == -1) {
         perror("Failed to create a socket");
         exit(EXIT_FAILURE);
       }
       memset(&sa, 0, sizeof sa);
       sa.sin_family = AF_INET;
       sa.sin_port = htons(1100);
       res = inet_pton(AF_INET, IP,
```

```
        &sa.sin_addr);
    connect(SocketFD, (struct sockaddr
        *)&sa, sizeof sa);

    Mat image;
    cap >> image;
    resize(image, image, Size(320,240));
    long imgSize =
        image.total()*image.elemSize();
    send(SocketFD,
        reinterpret_cast<char*>(image.data),
        imgSize, 0);
  }
  destroyAllWindows();
  shutdown(SocketFD, SHUT_RDWR);
  return EXIT_SUCCESS;
}
```