DIT 638 Cyber Physical Systems and Systems of Systems

Cybercar G5 Documentation

Group Members:

1. Nuria Cara Navas
2. Elsada Lagumdzic
3. Chi Hong (Nigel) Chao

Table of contents

1. Project Organization and Milestones [Nigel]

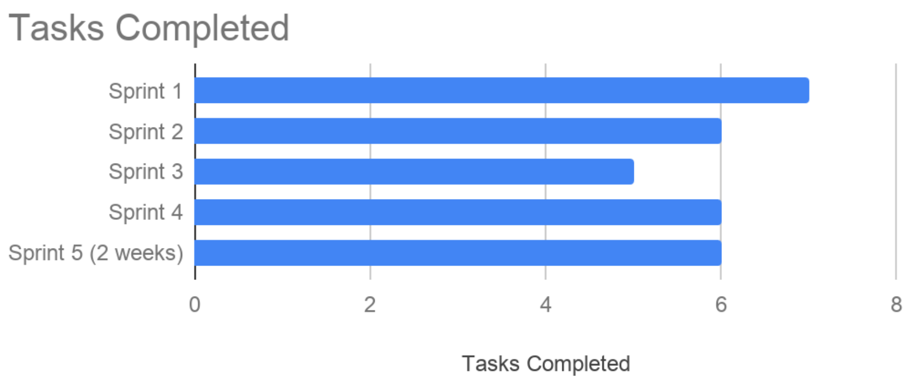
1.1 Project Organization

- 1 week sprints, where deliverables were planned to be met/given at the end of the sprint.
  - A sprint started on a Thursday and ended on the next Wednesday.
- On-site meetings around 4 days a week.
- Members gradually specialized in different areas due to time constraints.
- General project knowledge attempted to be disseminated across the members in meetings so that the group knew about the important concepts and ideas in the program.
- Trello was used to keep track of tasks, completed features, and prevent overlapping tasks.

1.2 Milestone Planning

Week (Sprint) 1	- Researched and implemented basic color tracking (no optimizations). - Quickly learned how to record scenarios and run them. - Wrote and improved on code of conduct
Week (Sprint) 2	- Kept group members on the same page in terms of knowledge acquired - Basic ACC with front sensor and shape detection running on car.
Week (Sprint) 3	- Began work on Stop Sign Detection using object detection. - Explored object recognition feasibility for the car. - Began work on following car using color/shape detection.
Week (Sprint) 4	- Project structure refactored to reflect the architecture we wanted. - Began testing ACC and Stop Sign Detection interacting together on car. - Recognizing other cars (not the leading car) in the intersection correctly. - Redesigned intersection and redefined more specific scenarios.
Sprint 5, 2 weeks	- More robust ACC (slower speed), color and shape detection (brightening video). - Implemented movements to leave the intersection. - Following a car until the stop line of the intersection more responsive and robust. - Began work on knowing when to leave the intersection after arriving at stop line.

Chart Displaying Number of Tasks Completed for each Sprint
A task was considered completed when it was reviewed by a group member and the feature was merged with master.



2. Implemented Algorithmic Concepts

2.1. Leading Car, Safe Distance Nigel

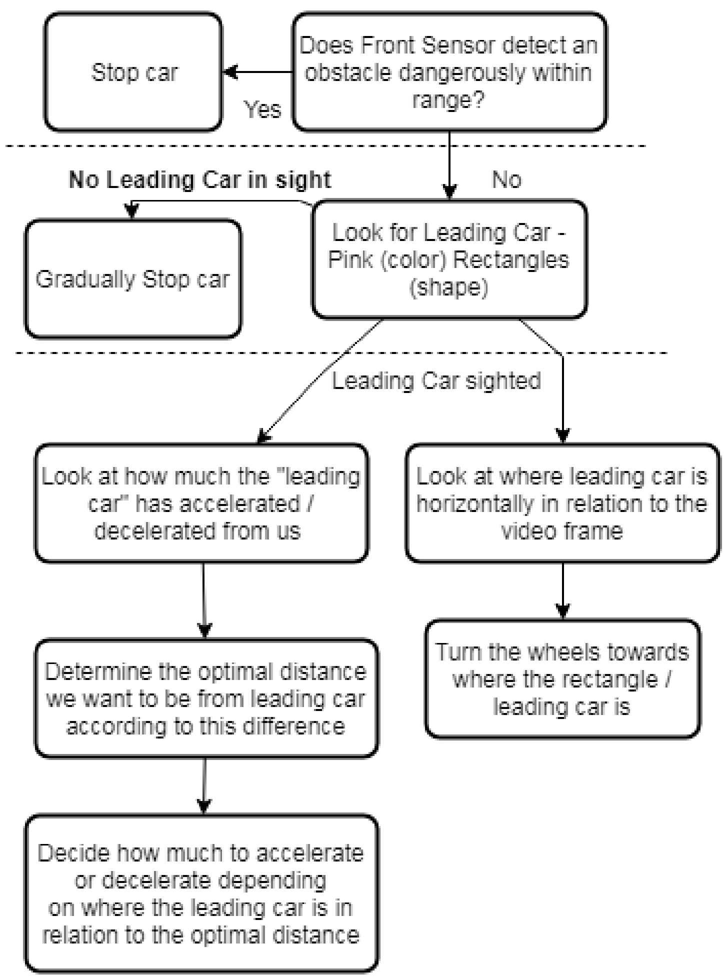

Stop car
Does Front Sensor detect an obstacle dangerously within range?
Yes
No Leading Car in sight
No
Gradually Stop car
Look for Leading Car - Pink (color) Rectangles (shape)
Leading Car sighted
Look at how much the "leading car" has accelerated / decelerated from us
Look at where leading car is horizontally in relation to the video frame
Determine the optimal distance we want to be from leading car according to this difference
Turn the wheels towards where the rectangle / leading car is
Decide how much to accelerate or decelerate depending on where the leading car is in relation to the optimal distance

Image recognition and sensor readings are used to follow a leading car. Color and shape recognition is used to detect a placeholder for the leading car. Color detection, done by filtering out pixel values below a threshold, is first used before shape detection, done by first scanning the frame for closed loop contours and checking if there are 4 right angles, as the inverse would fail to look for shapes of only the color we want.

For image recognition, we have used OpenCV as it was deemed feasible and sufficient for our needs. Furthermore, OpenCV gave us the possibility of using object detection in the future in case color was not robust enough. Finally, OpenCV is open source, making it easy to access.

If a leading car is in sight, speed and steering corrections are given to follow it. If no leading car is in sight, the car will gradually stop. For illustration purposes, the diagram on the left shows both speed and steering being processed in tandem, but it should be noted that speed is processed first.

A safe distance is calculated depending on how much the leading car has accelerated or decelerated from our car by comparing the previous and current areas of the bounding boxes. Depending on where the leading car is in relation to the safe distance, the car decides how much speed to add or reduce.

A steering correction is calculated depending on where the horizontal center of the leading car lies on the frame. The error of the current horizontal center and the frame center is calculated and then translated to a steering value.

To add an extra layer of safety, an “Emergency Brake” is implemented. If the front sensor senses an obstacle that is too close, the car will stop regardless of how far away the leading car is.

2.2. Controlling of the vehicle movements Elsada

Vehicle’s movement is dynamic and varying from scenario to scenario in which the car is used. However, car movements within the scenarios can be clearly divided into several parts where the car behavior is more consistent and static when looking isolate. This implies that by applying the technique “Divide and conquer” it is possible split the logic of control of the car movements onto different parts that can interact with each other.

To achieve this idea, the control of the car movements is spread across the whole software architecture which consists of several microservices that communicating between each other by exchanging messages via UDP multicast. Via those messages other microservices influence the way car moves. Each defined message is a part of the algorithm for control of car movement. Messages are defined according to its purpose in defined scenario.

Car movements are managed in one separate microservice primarily dedicated to listen to the messages from other microservices in our project and then manage the car movements according to messages. Messages are some sort of trigger for certain car reaction such as moving forward, steering or stopping and all combined for a certain scenario. Prioritization of messages is solved by checking different states of the software within scenarios, or more precisely by adding conditions. For example the car will not react on a message to approach the stop line if before that it has not been still for certain period of time, which means that some other scenario needed to happen first. Once it receives the message for certain movement, the Move Car will apply appropriate logic and forward it to the car, via messages to Beagle Bone. More specific car movements are described within scenarios such as Acc, Stopping at the stop line, dealing with intersection and similar.

2.3. Stopping When Approaching the stop line at Intersection $^{\text{Nuria}}$

For stopping before the stop line once the car is approaching the intersection, we are using the raspberry Pi camera and the object detection of the stop sign. We assume that the vehicle in front of us has just left the intersection. At the same time a stop sign is being recognize from the position our car is waiting to receive the next instructions. When the leading car is not seen for a certain amount of time, a message will be sent from accSafeDistance to moveCar microservice, triggering our car to start moving forward at a low speed towards the stop line. The moment the camera stops recognizing the stop sign, a message will be sent to moveCar (stopSignRecognized to moveCar) in order to stop the car from continuing going forward, as the disappearance of the stop sign means that we are close enough to the stop line at the intersection.
The disadvantage of this is that we do not know with certainty that the car will stop on the correct spot every time. It's hard to reproduce good results and we depend on other factors, such as the distance and the angle where the car was previously stop. Sometimes it will recognize that the stop sign has disappeared and will stop too far away or way too close to the stop line.

2.4. Determining When to Leave Intersection $^{\text{Nigel}}$

Color and shape recognition, in a similar way described in the ACC section above, is currently used to look for cars in the intersection. If the trained classifier is robust enough, we are considering to transition to object recognition. Other cars begin to be detected and counted after the leading car has left, and begin looking for cars leaving after the car arrives at the stop line. It is assumed that any car detected is in front of us in the queue. After arriving at the line, the car will. The previous position is compared to the current one, and depending on the trajectory, we assume a car is leaving towards a certain direction. When a car leaves the frame, a car is deducted from the number of cars before ours. Use of sensors have been considered to detect leaving cars, and use of both image recognition and sensors would be most ideal. When no cars are left in the queue, the car is ready to leave.

2.5. Leaving the Intersection $^{\text{Elsada}}$

From conceptual algorithmic aspect, leaving the intersection implies the steps the car does once it know that it is its turn to cross the intersection and it is safe to move. In the moment of writing this document, the scenario for leaving the intersection is not completely implemented so we will try to explain how it should look like, with reserving the right for eventual changes. For this intersection scenario to happen, we assume the car will be standing straight on the stop line.

However, leaving the intersection shall start only after the car gets a message saying that it it its turn. So, once that message is received, the program will ask the user to input on the console the direction in which the car should go on the intersection.The user should have possibility to input number 1 for turning right, number 2 for going straight or number 3 for turning left. Once entered, the input should be checked to see if the required turning is allowed. If not, the program will ask for new input, and if yes, the program will forward a message to the car asking to move in direction required in the input. The car will then move forward and steer to turn, then after turning straighten the wheels and continue moving forward a little bit, and then stop.

2.6. Stop sign object detection $^{\text{Nuria}}$

For the object identification of the stop sign, the Haar cascade algorithm has been used with an already trained classifier. The area of the boundary box is used to start the identification of the stop sign from a certain distance, as we only want to start recognizing it after the vehicle in front of us has stopped, in order to avoid that the stopSignRecognition microservice sends a stop sign out of sight message to moveCar microservice before we start moving forward to the stop line, as this will make the moveCar microservice stop listening for other messages incoming from another microservices. So once the area is big enough, a boolean will be sent, meaning that a stop sign has been recognized. If there is no stop sign, a false will be sent instead.

This algorithms only handles detection of a stop sign and sends messages weather it was recognize or not, by checking and comparing previous with current frames to make the identification more robust and avoid false positives.

2.7. Car object detection $^{\text{Nuria}}$

In order to be able to detect other cars at the intersection, object detection using haar cascade classifier was implement. The car object detection will send a message weather a car has been identified or not. For this our own classifier was trained using positives and negatives pictures of the kiwi miniature car that were taken using the raspberryPi camera. The reason we used the raspberry camera and not a mobile phone camera was that the mobile camera had better picture and video resolution, which made the classifier not robust in identification. It created a lot of false positives.
The messages will be sent to carQueu microservice in order to add them into the queue.

The reason we decided to implement car object detection instead of color detection for the other cars at the intersection was that object detection provides a more robust detection and depends on less factors (such as light condition shadows, etc) that can affect the result of detecting the right object and the right amount of cars. It would also make possible an easier implementation of object tracking, which will allow to follow when the vehicles leave the intersection.

However, this feature might not be included in the final product, as we could not make it robust and reliable enough in order to identify the other cars on the different angles. The trained classifier is only able to identify the back of a car, but it does not work for lateral or front view of the car at the moment.

3. Detection and Movement Algorithms

3.1. Leading Car, Safe Distance[1] Nigel

To provide higher reliability in detecting the correct shape and color, the incoming video feed is brightened[2] before the color and shape detection is done. This slightly improves the detection rate and helps reduce unwanted environmental variables that we are unable to control, such as shadows. This brightening is done by plotting the pixel values of the grayscale frame in a histogram, then averaging out the current range.

Shape detection[3], in our case rectangles, are detected by first downscaling and then re-upscaling the frame to filter out the noise. Edge detection is then used, and each contour is stored in an array. Contours are then closed, and only those with 4 sides and of a reasonable size are considered. Finally, the most acute angle is checked to see if it is close enough to a square.

When a leading car is detected, a PID(Proportional-Integral-Derivative)[4] controller is used to follow it. However, only the Proportional(P) part is used here for both speed and steering as it was considered to be robust enough. The P-controller adjusts speed and steering depending on where the leading car is in relation to a setpoint. The area of the bounding box is used to determine the distance between our car and the leading car, while the center point of the box is used for the horizontal center.

To prevent overshooting, the previous area of the leading car is compared against the current area in order to know whether the car in front has accelerated or decelerated, effectively predicting what will happen next. By knowing whether the car in front is closing in or going farther away, adjustments to the safe distance is made to react earlier and maintain a good amount of distance from the leading car. The greater the distance, the farther the safe distance will be. This will result in earlier braking and softer acceleration. When the leading car distance is maintained, the car slowly decelerates to better maintain distance.

The horizontal center of the detected bounding box is subtracted from the frame center to determine the horizontal position of the leading car. The value is then translated to an absolute steering angle for the car.

It should be noted that although both speed and steering use P-controllers, they are used differently. For speed, the calculated value is added upon the current speed, while the steering value replaces the current direction entirely. This was done due to the different nature of the Kiwi car's speed and steering. A base initial amount was required to propel the car forward, but the servos was more responsive to slight alterations of values.

To reliably control the car after detecting these corrections, several checks and limits are placed on the car. A starting and maximum speed is given to kickstart the car and prevent the car from going too fast respectively, and steering values are limited due to hardware limitations. Losing visual of the leading car also slows down the car by reducing speed each frame, and the car decelerates only if the car is fast enough. Safety checks such as the front sensor acting as an emergency brake help make braking more robust in case the camera may fail to detect braking in time.

[1] See Fig.1 for a sequence diagram.
[2] Refer to 4) in References of Algorithmic Details.
[3] Refer to 3) in References of Algorithmic Details for more details.
[4] Refer to 5) in References of Algorithmic Details.

3.2. Controlling of the vehicle’s movements in the Move Car Elsada

Move Car microservices is primarily dedicated to listen and interpret messages from other microservices and manage the car movement in scenarios such as ACC, stopping on the line and moving on the intersection. Each message is defined in the messages.odvd file and has its unique ID that helps differentiate messages in the same communication channel. Messages are handled in its own handler (in the Move Car - Follow.cpp) which in turn helps to separate and distinguish scenarios and car movements. Handlers are used for processing the input message and to apply decision logic for car movement according to the message ID. Depending on the scenario the car movements can include for example moving forward, steering, stopping and counting the time for which the car was stationary. However, all these movements are triggered by messages from other microservices. Such messages usually carry the interpretation of the data gained and processed by detection algorithms. This also means that the robustness and reliability of the car movements largely depend on robustness of other microservices such as ACC distance and Stop sign detection.

3.3. Stopping When Approaching the stop line at Intersection Nuria

Parting from the scenario where the moveCar microservice has already received a message from AccSafeDistance when the leading car is not being seen anymore after five seconds, the car will start moving forward at a speed of 0.11.
To be able to stop the car before the stop line, the StopSignRecognition microservice will communicate and send a message to moveCar. This message will only be sent when the stop sign is not being recognized anymore afterwards it was previously recognized, as well as when the area of the stop sign is big enough. The moveCar will stop the car once the message from the StopSignRecognition has been received.
This communication is possible by a defined message with an unique Id and identical naming on the odvd files in both microservices. This means that successfully stopping the car before the stop line depends mostly on the robustness of the stop sign recognition and on how fast the message is being sent and received between both of the microservices.

3.4. Determining When to Leave Intersection Nigel

Similar to ACC, the video feed is also brightened before color and shape recognition is done. For each detected car, a bounding box is drawn over it and the center position is checked. To know which lane the detected car was in, the frame is separated into 3 sections. According to which section the position fell on, the position is stored in a specific index of an array. The cars are counted depending on the array. When looking for cars leaving the intersection, the difference between the current and previous center positions of the bounding rectangles decide what direction the car is leaving. In the same context, if the current center is on the edge of the frame as well, it is considered that the car has left the intersection. For a car leaving at 12 o'clock, the car is considered to have left if smaller than a specific area. A timeout of 8 seconds is given to prevent false positives once a car has left, and the number of cars in queue is decreased by one. When the number of cars in queue hits 0, a message is sent to leave the intersection.

3.5. Leaving the Intersection Elsada

Leaving the intersection algorithm requires the usage and interaction of multiple microservices. As mentioned in the section 2.2 the car movements (turning) will be handled by one microservice (Move Car) while the input will be managed by a different microservice called Input Directions. The Input Direction is supposed to receive the message “your turn to move”, from the Car queue microservice when that microservice decides that it is our turn to leave the intersection. Once that message is received, the Input Direction will, via console, ask the user for the input. That input will be a trigger for a new message containing direction, which Input Direction sends to the Move Car. Once the Move car receives the direction, it will then set the speed and steering and combine them with delays to turn properly.

3.6. Stop sign object detection Nuria

In order to make the object detection robust and avoid false positives and false negatives, a method comparing the previous frames with the current frames that the raspberryPi camera is seeing has been implemented. This method can compare the specified amount of frames (the greater the amount of frames to check, the longer the response will be, causing a delay in when to stop the car) and it will check weather the it has seen a stop sign in at least a certain amount of frames or not. Once we know if there is a stop sign or not in front of us, this data will be used on the stop sign detection method to send the correct message to the moveCar. The message is defined on the odvd file as well as the unique Id in order to send the message to the right microservice.

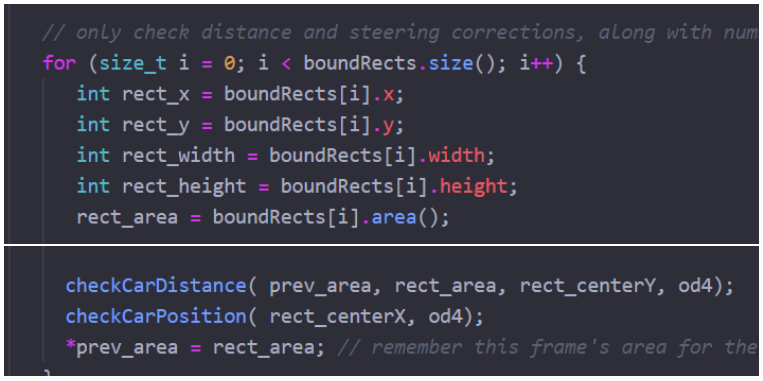
3.7. Car object detection Nuria

In order to train the classifier that will allow the detection of the cars, positive (images containing the car) and negative (images where there were no cars present) pictures of the kiwi miniature car were taken using the raspberryPi camera. Afterwards, the positive pictures were tagged with the location of where the car was located on each picture and how many cars did the picture contained. Each positive picture only contained one car, although the best would have been to take also pictures were more than one car was present, in order to make the classifier have a more robust detection. After all the positive pictures were tagged individually, a vec file has to be created to make it compatible for the opencv train cascade classifier object detector built-in function to start the training. Then the file containing the negative pictures and the vec file containing the positives ones, will be load into the opencv cascade classifier trainer function and the training will begin for twenty stages.

Then the created classifier needs to be put into the the same folder as the source code and copied into the docker file manually (specifying the location of the .xml file). Then load the input video into grayscale mode. If cars are found in the frames of the Pi camera, it will return the position of the detected cars and afterwards creates a ROI for the car detection. Finally, after detecting the cars, a message will be sent to carQueue microservice saying that a car has been recognize at the intersection.

4. Implementation and Source Code Details

4.1. Leading Car, Safe Distance Nigel

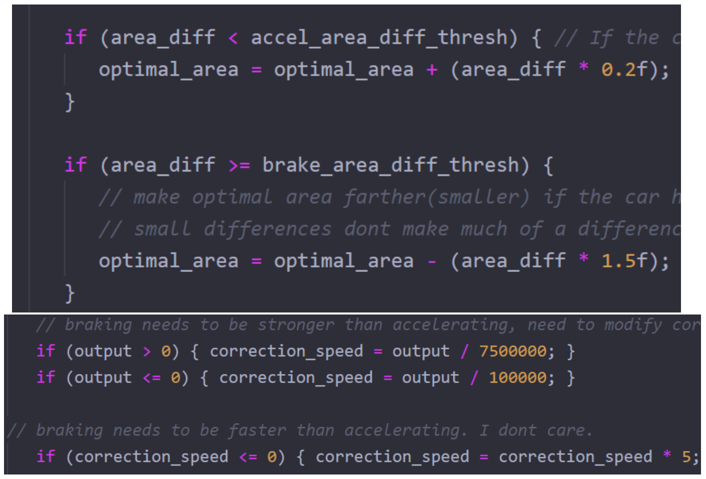


```
// only check distance and steering corrections, along with num
for (size_t i = 0; i < boundRects.size(); i++) {
    int rect_x = boundRects[i].x;
    int rect_y = boundRects[i].y;
    int rect_width = boundRects[i].width;
    int rect_height = boundRects[i].height;
    rect_area = boundRects[i].area();

  checkCarDistance( prev_area, rect_area, rect_centerY, od4);
  checkCarPosition( rect_centerX, od4);
  *prev_area = rect_area; // remember this frame's area for the
```

ACC involves 2 microservices, "accSafeDistance" and "MoveCar", working with each other. When a "car" is detected in "accSafeDistance", a bounding rectangle is created for each rectangle. Since the detected rectangle can be skewed, this ensures that each rectangle is upright, allowing us to obtain accurate measurements. For each box, the distance and position is checked. It is important to note that a pointer is used to remember the last frame's area.

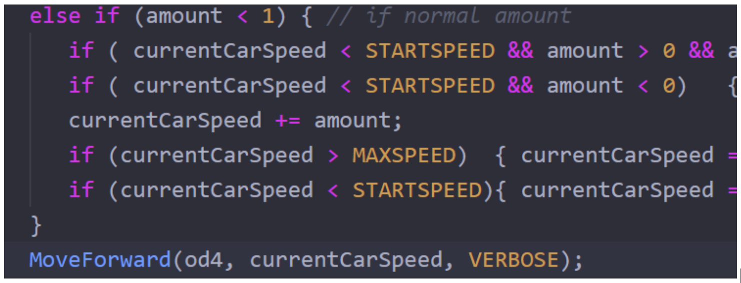
The code on the right shows how the safe distance is first adjusted before the error is calculated. It is interesting to note that the optimal area increases even if the car in front accelerates, but only slightly. This was done to further dampen acceleration as it was deemed too aggressive.

Due to multiple problems with overshooting when braking, it was decided that braking had to be **375** times harder than acceleration after testing. This is because the car has much fewer frames to react when the car is closing in in contrast to going away.



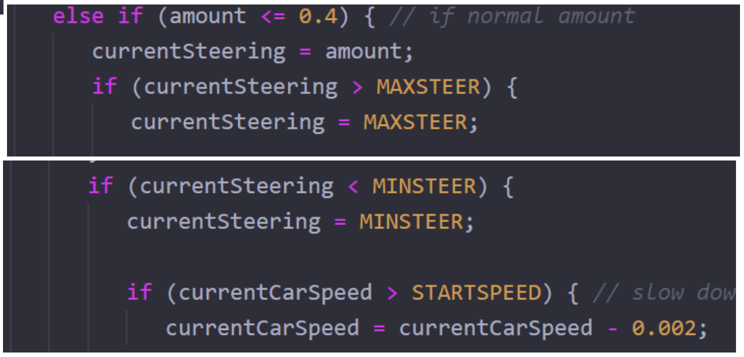
```
  if (area_diff < accel_area_diff_thresh) { // If the c
    optimal_area = optimal_area + (area_diff * 0.2f);
  }

  if (area_diff >= brake_area_diff_thresh) {
    // make optimal area farther(smaller) if the car h
    // small differences dont make much of a differenc
    optimal_area = optimal_area - (area_diff * 1.5f);
  }
  // braking needs to be stronger than accelerating, need to modify cor
  if (output > 0) { correction_speed = output / 7500000; }
  if (output <= 0) { correction_speed = output / 100000; }

// braking needs to be faster than accelerating. I dont care.
  if (correction_speed <= 0) { correction_speed = correction_speed * 5;
```

The code above resides in MoveCar, which interprets the speed and steering corrections, reacting accordingly. MoveCar listens to speed correction messages coming from accSafeDistance and as shown here, adds the correction to its existing amount.



```
else if (amount < 1) { // if normal amount
    if ( currentCarSpeed < STARTSPEED && amount > 0 && a
    if ( currentCarSpeed < STARTSPEED && amount < 0)   {
    currentCarSpeed += amount;
    if (currentCarSpeed > MAXSPEED)  { currentCarSpeed =
    if (currentCarSpeed < STARTSPEED){ currentCarSpeed =
}
MoveForward(od4, currentCarSpeed, VERBOSE);
```

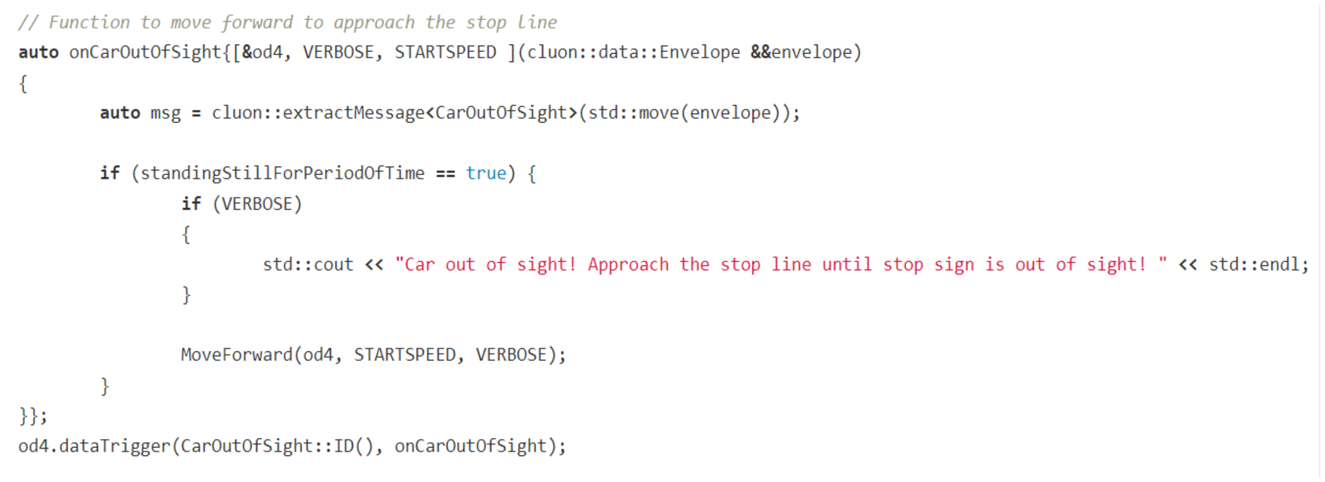
In contrast, as shown on the right, steering corrections replace the current value for maximum responsiveness. It is also interesting to note that the car is slowed down when the car turns to its maximum value. This was done to reduce blurriness as we realised that the resolution and blurriness prevented the car from recognizing squares at all.



```
else if (amount <= 0.4) { // if normal amount
    currentSteering = amount;
    if (currentSteering > MAXSTEER) {
        currentSteering = MAXSTEER;

    if (currentSteering < MINSTEER) {
        currentSteering = MINSTEER;

        if (currentCarSpeed > STARTSPEED) { // slow dow
            currentCarSpeed = currentCarSpeed - 0.002;
```

4.2. Implementation of the Move Car microservice $^{\text{Elsada}}$

Microservice managing the car movements is implemented through multiple callback functions acting as handlers for messages. These callback functions receive and process messages from other microservices and then send commands to the car via Pedal Request and Ground Steering request messages. Pre-installed microservice called opendlv-device-kiwi-prugw listens for pedal and steering messages to interface with cars motor and servo for movements. Callback functions are triggered by incoming messages with specific IDs. So the Move Car implements callback functions for ACC scenario such as OnSpeedCorrection, OnSteeringCorrection, OnStopCar, OnFrontDistance, OnCarOutOfsight, but also functions for other scenarios such as OnChooseDirection.



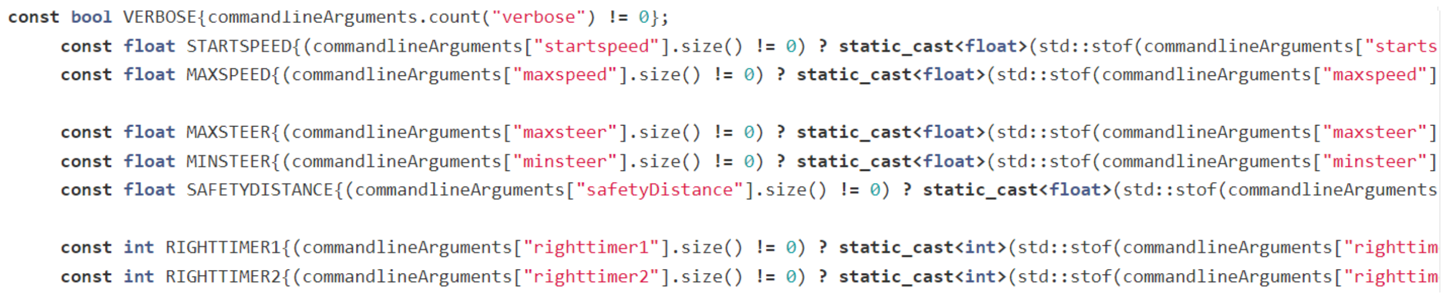
```
// Function to move forward to approach the stop line
auto onCarOutOfSight{[&od4, VERBOSE, STARTSPEED ](cluon::data::Envelope &&envelope)
{
        auto msg = cluon::extractMessage<CarOutOfSight>(std::move(envelope));

        if (standingStillForPeriodOfTime == true) {
                if (VERBOSE)
                {
                        std::cout << "Car out of sight! Approach the stop line until stop sign is out of sight! " << std::endl;
                }

                MoveForward(od4, STARTSPEED, VERBOSE);
        }
}};
od4.dataTrigger(CarOutOfSight::ID(), onCarOutOfSight);
```

The code snippet above shows the implementation of one of the callback functions.

Once the callback function is triggered, the message will be extracted from the Envelope with help of Cluon. According to the content of the message the car will react with certain movements like speed and steering corrections needed for the ACC. Handling the messages in callback functions allows their independent processing, as well as easier handling by splitting the code into several functions. But on the other hand, due to dependencies between handling of different messages and their prioritization, this approach requires the usage of global variables. Global variable are keeping the states of the software to know which part of the scenario was executed.

One of the key functions in the Move Car is the MoveForward. This function sets the speed for pedal position to move the car. Besides that, in the MoveForward function we also count the time for which the car was standing still (assume: standing behind followed car at the intersection). For counting the time we are using the c++ Chrono library. The function starts measuring the time when the speed of the car changes from a non-zero value to zero and only if was not already still for certain period of time. Measuring is always trigger from OnSpeedCorrection callback.

In order to avoid constant building of Docker images, for deployment, the Move Car microservice implements the possibility to forward CLI arguments (Command line interface) to our running application.This is shown in the following code snippet and it largely facilitated the testing.

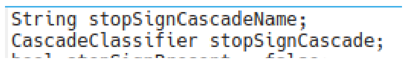


```
const bool VERBOSE{commandlineArguments.count("verbose") != 0};
    const float STARTSPEED{(commandlineArguments["startspeed"].size() != 0) ? static_cast<float>(std::stof(commandlineArguments["starts
    const float MAXSPEED{(commandlineArguments["maxspeed"].size() != 0) ? static_cast<float>(std::stof(commandlineArguments["maxspeed"]

    const float MAXSTEER{(commandlineArguments["maxsteer"].size() != 0) ? static_cast<float>(std::stof(commandlineArguments["maxsteer"]
    const float MINSTEER{(commandlineArguments["minsteer"].size() != 0) ? static_cast<float>(std::stof(commandlineArguments["minsteer"]
    const float SAFETYDISTANCE{(commandlineArguments["safetyDistance"].size() != 0) ? static_cast<float>(std::stof(commandlineArguments

    const int RIGHTTIMER1{(commandlineArguments["righttimer1"].size() != 0) ? static_cast<int>(std::stof(commandlineArguments["righttim
    const int RIGHTTIMER2{(commandlineArguments["righttimer2"].size() != 0) ? static_cast<int>(std::stof(commandlineArguments["righttim
```

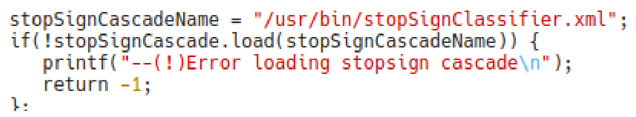
4.3. Implementation of stop sign object recognition microservice Nuria

The stop sign microservice is using the haar cascade algorithm for facial recognition already built in opencv. The algorithm has been adapted to be able to recognize stop signs instead of faces.

To be able to recognize the stop sign with the camera, the trained XML classifier needs to be loaded and then load the input video in grayscale mode.

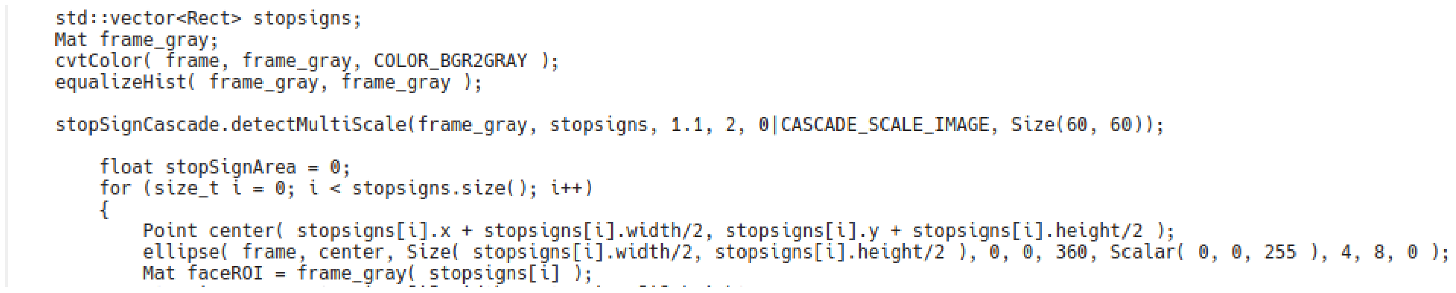


```
String stopSignCascadeName;
CascadeClassifier stopSignCascade;
```



```
stopSignCascadeName = "/usr/bin/stopSignClassifier.xml";
if(!stopSignCascade.load(stopSignCascadeName)) {
   printf("--(!)Error loading stopsign cascade\n");
   return -1;
};
```

After that, we find the stop signs in the video. If they have being identified, a position of the detected stop signs will be returned as Rect(x,y,w,h). After getting the locations a ROI will be created and applied for the stop signs. The boundary box has been changed to a circle for aesthetic purposes, but doesn’t affect the performance or the recognition of the stop sign in any way.



```
    std::vector<Rect> stopsigns;
    Mat frame_gray;
    cvtColor( frame, frame_gray, COLOR_BGR2GRAY );
    equalizeHist( frame_gray, frame_gray );

    stopSignCascade.detectMultiScale(frame_gray, stopsigns, 1.1, 2, 0|CASCADE_SCALE_IMAGE, Size(60, 60));

        float stopSignArea = 0;
        for (size_t i = 0; i < stopsigns.size(); i++)
        {
            Point center( stopsigns[i].x + stopsigns[i].width/2, stopsigns[i].y + stopsigns[i].height/2 );
            ellipse( frame, center, Size( stopsigns[i].width/2, stopsigns[i].height/2 ), 0, 0, 360, Scalar( 0, 0, 255 ), 4, 8, 0 );
            Mat faceROI = frame_gray( stopsigns[i] );
```

To make the algorithm more reliable and avoid false positives that could interfere with the stop at the stop line scenario, a comparison between the last nine frames and the frames currently being seen by the PI camera has been implemented. These frames will be saved on an array of booleans, which will contain true or false whether the stop sign has been identified or not. After looping through the array, if five of the booleans has been the same consecutively, then that will mean that there is a value to be reported. That value will be used by the ‘detectAndDisplayStopSign’ method on ‘stop-sign.cpp’ in order to know when to send the message correct message.

```
int noOfFramesWithStopsigns = 0;
//Loop over the array and collect all the trues.
for(int i = 0; i < lookBackNoOfFrames; i++) {
    if(seenFrameStopsigns[i]) {
        noOfFramesWithStopsigns++;
    }
}
if(noOfFramesWithStopsigns < NO_OF_STOPSIGNS_REQUIRED) {
    return false;
}
else {
    return true;
}
```

If the value to report is true, it will send a message saying that a stop sign is being detected. On the other hand, if the value to report is false, then the message that will be sent to moveCar microservice will be a false. MoveCar will only start listening to the messages sent by stopSignRecognition when there is not leading vehicle in front of us after following it to the stop line. Furthermore, the stopSignRecognition microservice will only start when the area of the stop sign is big enough, as recognizing the stop sign when we are far away from the intersection is unnecessary and could cause interference or send the messages before the cars has arrived at the stop line (as for example, follow until the stop line scenario).

```
bool valueToReport = insertCurrentFrameStopSign(stopSignArea > 5000);
if(stopSignPresent != valueToReport){
    stopSignPresent = valueToReport;
    stopSignPresenceUpdate.stopSignPresence(valueToReport);
    if(valueToReport) {
        std::cout << "sending stop sign detected message: " << std::endl;
    } else {
        std::cout << "sending NO stop sign present message: " << std::endl;
        od4->send(stopSignPresenceUpdate);
    }
```

By this we will assure that when moveCar start moving forwards and it loses the visual contact of the stop sign, it will mean that we are really close to the stop line and our vehicle will stop right before it.

4.5. Implementation of the scenario for leaving the intersection Elsada

Turning left, right or moving straight is implemented with hard coded values for speed, steering and delays. The steering is set to max value.

4.6. Implementation of the vehicle object recognition Nuria

This microservice has been worked on but unfortunately not yet implemented, as we were not able to make it robust enough to recognize the kiwi car on all the angles before the final product. The implementation and the logic behind is similar as previously described for the stop sign, but with a different classifier and sending other types of messages to the carQueue microservice. To avoid similar issues with false positives and false negatives already encountered on the stop sign recognition, another method including a boolean array of trues and falses comparing the current frames with the last ten ones, was implemented.

Some of the logic would have been changed to better adapt to the queue situation.

5. System Architecture Nuria

Our system architecture is based on microservice[5] architecture and the communication between the different ones via messages. These microservices are its own structured application which collects different services providing, in this way, low coupling and independent deployability.

In order to load and run these independent microservices in the kiwi miniature car, they need to be compatible with the hardware components. To be able to do this, the microservices need to be created using Docker. A more detailed explanation about Docker, microservices and hardware/software integration will be made on the Hardware and software integration section[6].

The software architecture consists of three main microservices. The “move car”, the “acc safe distance” and the “stop sign recognition” microservice. The microservices that are thought of being included shortly are “Input Direction” and “car Queue”, as they are being worked on for before the final demo. The “don’t turn left sign recognition” might not be added by in the final product, as we have considered that this feature is not of high priority.

How the already implemented microservices interact with each other:

- The **moveCar** microservice handles the speed and the steering of the car as well as receiving readings from the sensor of the car. For speed and steering corrections the “moveCar” will send messages (“ opendvl.proxy.PedalPositionRequest” and “opendvl.proxy.GroundSteeringRequest”) to the “opendvldevice-kiwi” microservice (included in the BeagleBone pre-installed microservices) and this last microservice will react according to the messages received by interacting with the hardware of the car. Furthermore, “moveCar” will receive messages from “AccSafeDistance” and “stopSignRecognition” in order to successfully handle the follow the car until the stop line scenario. Possibly future microservices, such as InputDirection will also communicate with “moveCar” in order to take the chosen direction on the console.
- The **accSafeDistance** handles color and shape recognition as well as calculating the safe distance that our car should keep with the vehicle that’s in front of us, in order to be able to avoid collision. In the case that a vehicle has been recognized in front of us, this microservice will send the message of speed correction to the “moveCar” so it can start following it. Once started following, it will also send messages of steering correction in case the car in front of us takes different directions. If the our vehicle has being stopped for more than 7 seconds, then the message “car out of sight” will be sent to moveCar. MoveCar then will stop listening for messages from AccSafeDistance.
- **Stop sign Recognition** will only interact with the moveCar microservice by sending a boolean message whether a the stop sign has been detected or not. However, moveCar will only listen to the message sent by this microservice, if it has only previously received the message “car out of sight” from AccSafeDistance and it has been standing on the same position (with speed 0) for more than 7 seconds. The message that the stopSignRecognition will send to moveCar

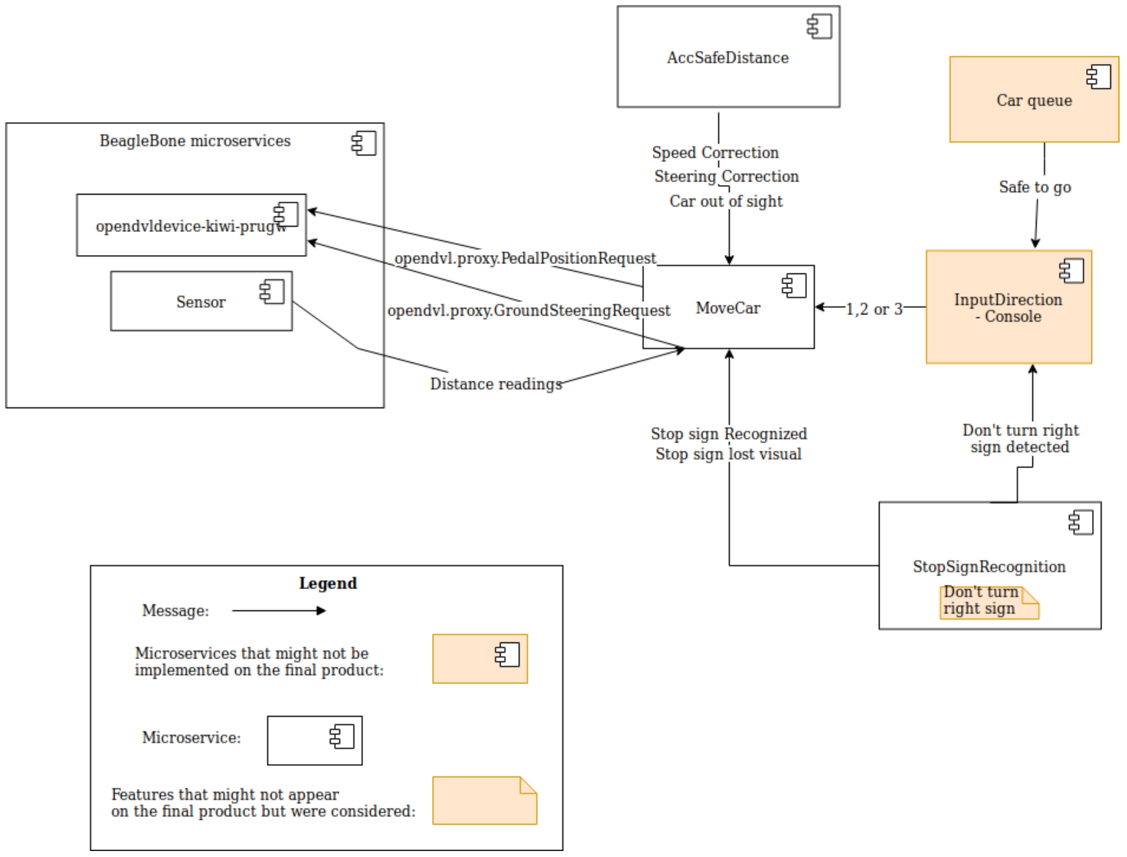
[5] “Microservice architecture”, Chris Richardson, 2018. https://microservices.io
[6] “About images, containers, and storage drivers”, Docker Docs, guides., 2017 https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers

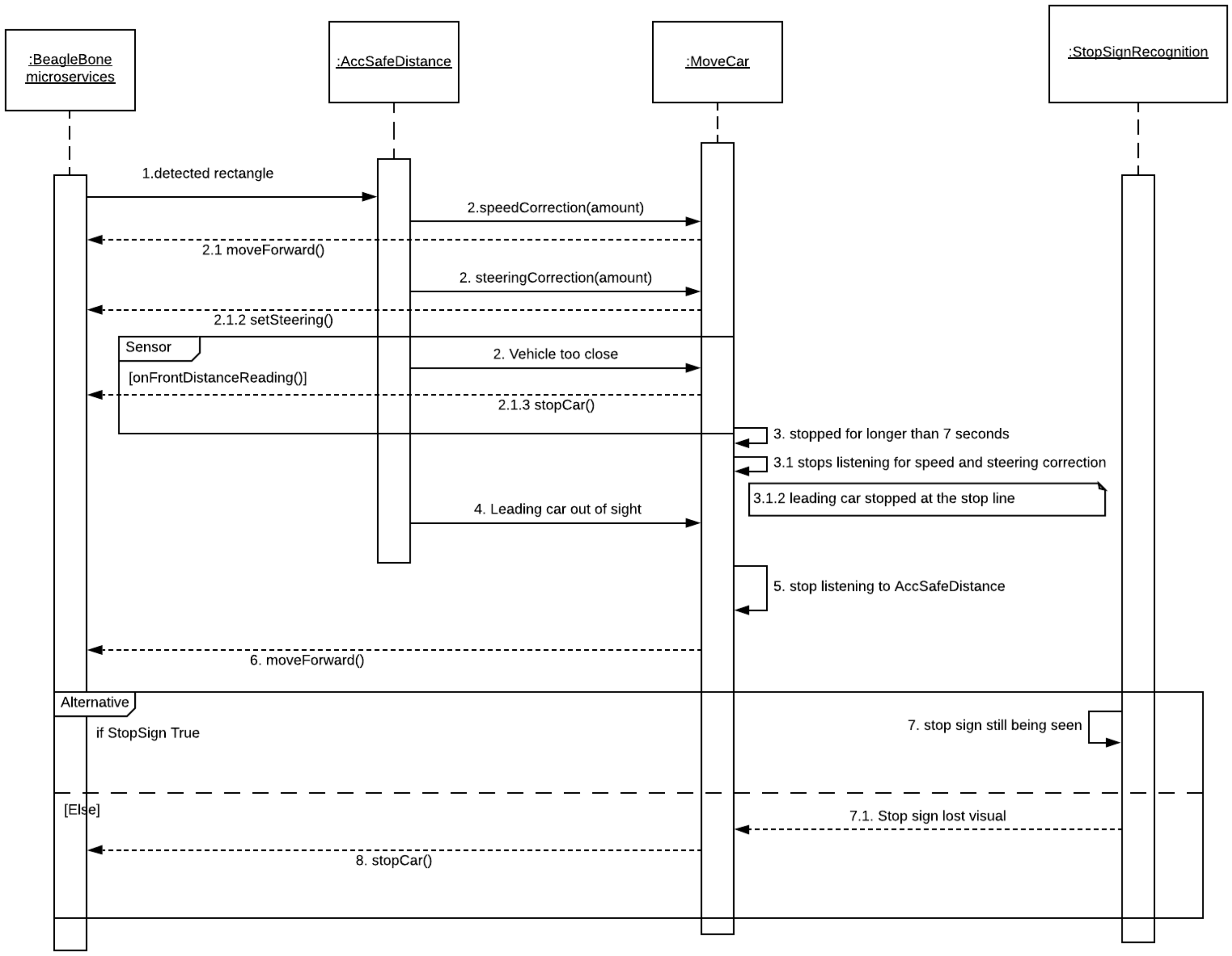
will be “Stop sign lost/Stop sign not being seen anymore”, which means that we are close the stop line.

Possible future microservices:

- **Input Direction**: Once the moveCar has received the message from the carQueue microservice that we are safe to go at the intersection (meaning we are next in the queue), the input direction will ask the user to enter a number such as one (turn right), two (straight) or three (turn left) manually in the console. Once the direction has been chosen, the inputDirection will send the message to moveCar to take the direction chosen. However, if there is an existing “don’t turn right” sign that has been previously recognized, then even thought the selected direction was ‘1’, the inputDirection should notify the user with a message on the console, that the chosen direction can’t be taken as it’s forbidden on the intersection, and won’t send the message to moveCar.
- **Car Queue**: This microservice is responsible of informing the moveCar microservice in the correct time in order to handle correctly and safely the intersection. This can be achieved by tracking the other cars that arrived at the intersection before us. After notifying to the moveCar microservice that it is safe to go, carQueue is also responsible to communicate with the InputDirection microservice in order for the car to take a direction.

The following component diagram and sequence diagram will help to better clarify the how the messages are handled and how the different microservices interact with each other:




:BeagleBone microservices
:AccSafeDistance
:MoveCar
:StopSignRecognition
1.detected rectangle
2.speedCorrection(amount)
2.1 moveForward()
2. steeringCorrection(amount)
2.1.2 setSteering()
Sensor
2. Vehicle too close
[onFrontDistanceReading()]
2.1.3 stopCar()
3. stopped for longer than 7 seconds
3.1 stops listening for speed and steering correction
3.1.2 leading car stopped at the stop line
4. Leading car out of sight
5. stop listening to AccSafeDistance
6. moveForward()
Alternative
if StopSign True
7. stop sign still being seen
[Else]
7.1. Stop sign lost visual
8. stopCar()

Sequence diagram showing the communication between the different microservices on the “follow the leading vehicle until the stop line and stop at the stop line at the intersection” scenario.

6. Hardware and Software Integration Elsada

In order to explain the hardware and software integration, this section will first mention the pre installed components that came on the Kiwi, the miniature robotic vehicle, and then later on explain the way those components were used in our project!

The list of the hardware components[7] that came with the car and were used for this project can be found in the appendix. The key components for the Kiwi care, and therefore even for our project, are the two boards, Beagle Bone and Raspberry Pi. Those two boards are connected via the USB cable, and they deal with the software logic and car control. The boards provide Wi-Fi connection and in addition, the Beagle Bone also provide the Bluetooth connection. Sensors are indirectly connected to the Beagle Bone which also can engage and release the motors. Camera on the other hand is connected to the Raspberry Pi. Computer connection with the Kiwi car, or more precisely with its boards, is established via Wi-Fi that came preinstalled on the Kiwi car.

Software components are mapped into separate micro services that, according to their logical purpose, can be running on both the Raspberry Pi or the Beagle Bone. In this context, micro services represent an architectural style where an application is built as a set of small services /units running independently. This architectural approach improves modularity, facilitates continuous development and deployment, but at the same time it requires to carefully choose the support technologies including communication protocols.

So in order to support micro service architecture, we use Docker. Docker is a program that simplifies the deployment and management of micro services by building them into images. These images are the tar files which contain different layers of our application, including libraries and other dependencies needed for an executable code. Images are then created with the docker build command, and once started with the docker run command, they will produce containers. Containers are some sort of runtime instance of an image.

As mentioned, the Kiwi car can have micro services running on both boards[8]. On the Raspberry pi there are three pre installed micro services: the first one is dealing with the camera , the second one with the encoding of frames into opdlv messages and the third one is providing a web application for data view. Micro service on the Beagle Bone is mostly dealing with servo and motor by managing the car control and movements.

In term of communication protocols, the Kiwi car, and therefore even our project relies on UDP multicast protocol and OpenDLV environment, that allow communication between microservices. The

[7] "chalmers-revere/opendlv-tutorial-kiwi - GitHub." https://github.com/chalmers-revere/opendlv-tutorial-kiwi/tree/master/image-postprocessing-opencv-python. Accessed 12 May. 2019.
[8] Idem.

communication is done via messages between services belonging to the same UDP multicast session or more precisely to the same address marked as --cid. Messages are packed into Envelopes and can be exchanged even between services running on different boards, in this case Raspberry Pi and Beagle Bone. Those two boards are connected via usb cable which allows them to interact and uses each other's data.

For our our project we used almost all off above described, pre installed hardware and software components and, at the same time, we relied on their integration. This means, that the main hardware parts for us were the two boards, camera and front sensor, integrated with the software via OpenDlv environment and UDP, network cable and Docker, as described above. While using camera we also rely on pre installed micro services to get encoded frames which we then process in order to detect specific colors and /or objects. The front sensor is used as some sort of backup for emergency braking in cases when the car does not break enough relying on camera. Since we are using camera a lot, our docker images are mainly deployed on Raspberry Pi board which then communicates with Beagle Bone for the car control. The images on the car communicate via messages.

The following diagram shows a physical deployment of our project as well as the integration of the hardware and software. Preinstalled software components and not used hardware components will be shown compactly or omitted.

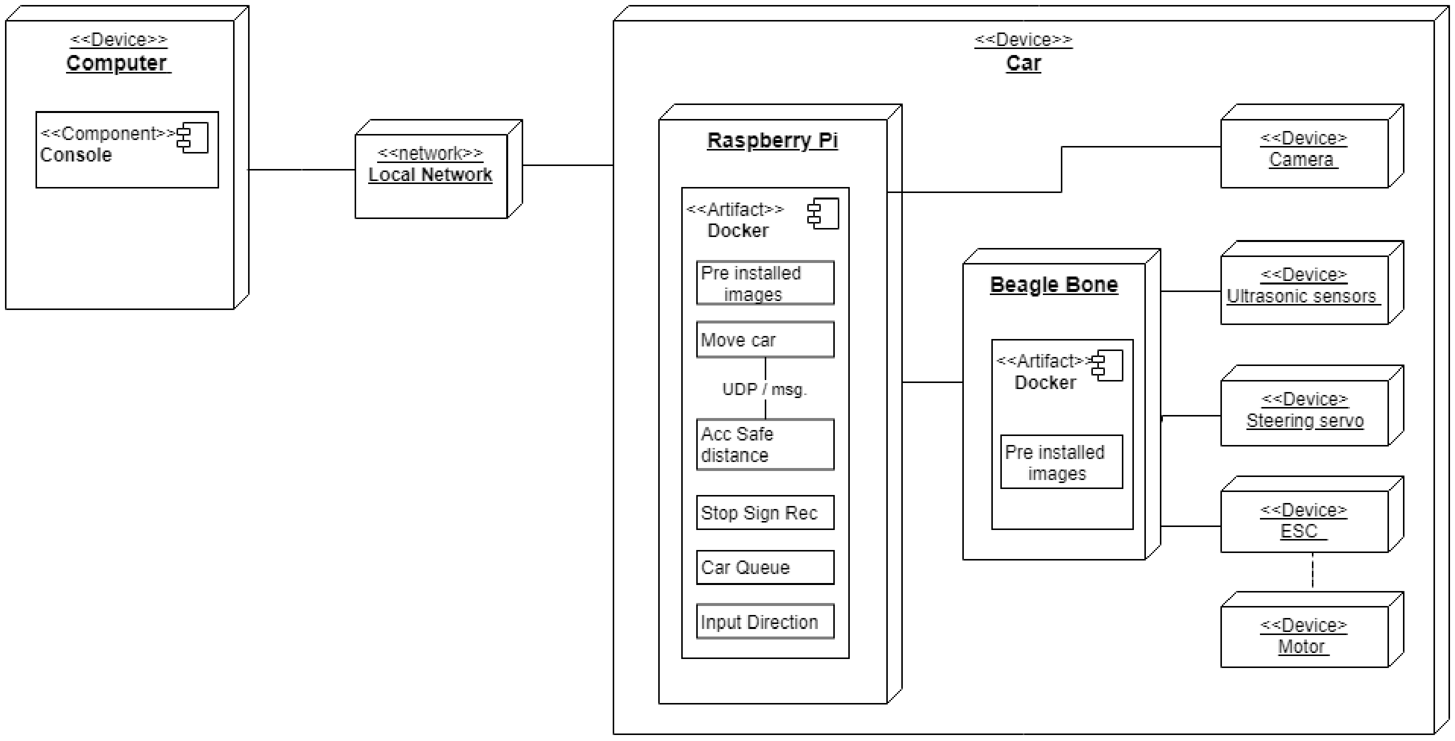


Diagram description:

The diagram shows two main nodes: the computer which controls and compiles part of software related to the project, and the car node where the main logic is deployed and executed. These two nodes communicate via local network (computer connects to the car’s Wi-Fi). Within the car node, reside multiple other nodes such as Raspberry Pi, Beagle Bone, Ultrasonic Sensors, Camera, ESC (Electronic Speed control) and motor. Infrared sensors are not part of the diagram since they were not explicitly used for the project. All these devices have interconnections and dependencies as described above. Connections are presented as a line and dependencies as a dashed line.

It is important to mention that the motor device depends to the ESC which is then connected to the Beagle Bone. According to the UML notation, the Docker is presented as an artifact (software product) which resides within the Raspberry Pi and Beagle Bone. The Docker itself contains images / containers. As shown on the diagram, the images / containers are connected via UDP and communicate via messages. For the sake of space, all the preinstalled images are shown as one component while the images created within our project are shown separately, as different components. The same applies for connections between images: the diagram shows only the communication between Move Car and Acc Safe distance but all images can communicate thanks to the same cid address.

7. Test Results and Descriptions Nigel

Multiple methods were used to ensure an acceptable level of quality. As the project progressed, so did testing, and some methods began to be more favored than others. Some methods of testing were unavailable depending on the functionality of the component. In particular, two types of functionality had different testing procedures: Image Recognition and Car Movement.

Regardless of functionality, the general process of integrating code into the hardware of the car consisted of 3 phases: Local, Recordings, and finally the On-site. Firstly, the program was run and tested locally. Recordings were then used to test whether or not it would compile on the car and behave properly in a recorded scenario. On-site testing was then done to not only make sure it would perform on the car, but also to spot possible cases in which the functionality would fail.

Local Testing

Local testing was particularly important during the early stages of development of a feature. Being the fastest in terms of testing iterations, testing locally allowed for easy experimentation. This was particularly important for image recognition - tuning was a big part of seeing what worked and what didn’t. Naturally, the Picamera and the webcam were different both in software and hardware, which meant that a functionality working locally did not mean that the car would behave the same.

Car movements were much more difficult to test locally. Initially, messages were tested to see if they could be received in two instances of the same program, but this was quickly outgrown - there was no need to test if the messages were received, but instead how the car reacted to the message.

One of the basic advantages of local testing was to ensure that the program compiled properly before moving on to more complex testing procedures such as on site testing. Ensuring that the program compiled was also the least a group member could do before accepting a merge request.

Recordings

Recordings were mainly used in the testing of image recognition as it car movements could not be “run” on them. As testing on recordings required Docker and similar steps with integrating code on the car, recordings ensured that it could at least compile on the car properly. Although one iteration of testing on recordings required slightly more time compared to local testing, it gave confidence that compiling the same code for the car would perform in a similar fashion. At the very least, it would run.

Testing code on recordings allowed us to see whether the program could execute the desired functionality on the car. This became particularly important when specific lighting and shadows would affect how the program would react in image recognition functionalities. The specific environment in which it was testing on would influence how the code would be tweaked.

On-Site

As the final step in the testing procedure, on-site testing was the most time consuming, but provided the highest amount of confidence that we had a certain degree of quality. Although on-site testing was sometimes done individually, other group members needed to be present if a functionality was to be approved, or to gauge the degree of success[9] of a functionality in a scenario. As on-site testing was performed in its intended environment, it alerted us about unconsidered aspects of the environment, and was the best way to determine the level of robustness our program had.

As car movements relied heavily on on-site testing, alternate methods to test in shorter iterations were required. One method we used was to have variable values be modifiable in the command line, which greatly improved the testing speed of car movements.

[9] Please refer to “Intersection Definition” and “Intersection Scenarios” in the appendix for the environment we tested on to measure success.

8. References for Algorithmic Details

1) “Cascade Classifier” openCV 2.4.13.7 documentation. OpenCV development team, 2011-2014.https://docs.opencv.org/2.4/doc/tutorials/objdetect/cascade_classifier/cascade_classifier.html
2) “Stopsigns, Detecting stop signs in Google Street View images of a provided address”, Markgaynor, 2018. https://github.com/markgaynor/stopsigns
3) “Datasets useful for Kiwi use cases”, olbender 2018. Chalmers-revere, opendlv-kiwi-data, kiwi detection folder. https://github.com/chalmers-revere/opendlv-kiwi-data/tree/master/kiwi_detection
4) “Square Detector”, Alexander Alekhin, Dmitriy Anisimov, Steven Puttemans, Suleyman Turkmen. OpenCV Examples. https://github.com/opencv/opencv/blob/master/samples/cpp/squares.cpp
5) “Auto Brightness and Contrast”, pklab, OpenCV Answers. https://answers.opencv.org/question/75510/how-to-make-auto-adjustmentsbrightness-and-contrast-for-image-android-opencv-image-correction/
6) “Microservice architecture”, Chris Richardson, 2018. https://microservices.io
7) “About images, containers and storage drivers”, Docker Docs, guides, 2017. https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers
8) “PID controller parameters”, Andy. https://robotics.stackexchange.com/questions/9786/how-do-the-pid-parameters-kp-ki-and-kd-affect-the-heading-of-a-differential
9) Libcluon Documentation, Christian Berger. https://chrberger.github.io/libcluon/index.html
10) Chalmers-revere/opendlv-tutorial-kiwi - GitHub. https://github.com/chalmers-revere/opendlv-tutorial-kiwi/tree/master/image-postprocessing-opencv-python. Accessed 12 May. 2019.
11) For the time counter:
https://en.cppreference.com/w/cpp/chrono/system_clock/now
https://en.cppreference.com/w/cpp/chrono

9. Project Retrospective The whole team

What worked well:

- We worked devotedly and quickly to meet the deadlines
- We learned about different technologies, but also about environment and hardware limitations and their impact on the software.
- We learned how to manage stressful situations when hardware issues were encountered and how to act accordingly.
- We tried to improve the mood and moral of the group by supporting each other when someone was having a hard time.
- We tried to deal with issues as fast as possible, making sure that the effects were as minimal as possible.
- Meeting frequently on site helped greatly in keeping others up to date on what was being worked on.

What did not work well:

- We didn't take the necessary amount of time to think and design the software at the beginning of the project. It would be easier to understand and implement scenarios if we had diagrams and descriptions.
- We did not structurally planned and clearly defined our testing approaches and testing scenarios
- Communication between the group failed a couple of times, resulting on logic change in already implemented features. However, we learned from this failures and became better at communicating with each other, so nobody is working on features that won't be implemented later on.
- Stand up meetings were quickly not followed, but informing others was fortunately not a problem due to the frequent face to face communication.
- We did not clearly define or set proper rules for the process of development from the perspective of the version control system (Gitlab) in the beginning. Although the general process was agreed on, we did not take advantage of the many features gitlab offered. For example, tags could have been used to track progress.

Appendix

Appendix of test protocols and charts where applicable (not counted to the students' individual contribution and also not counted to the overall document's page limit).

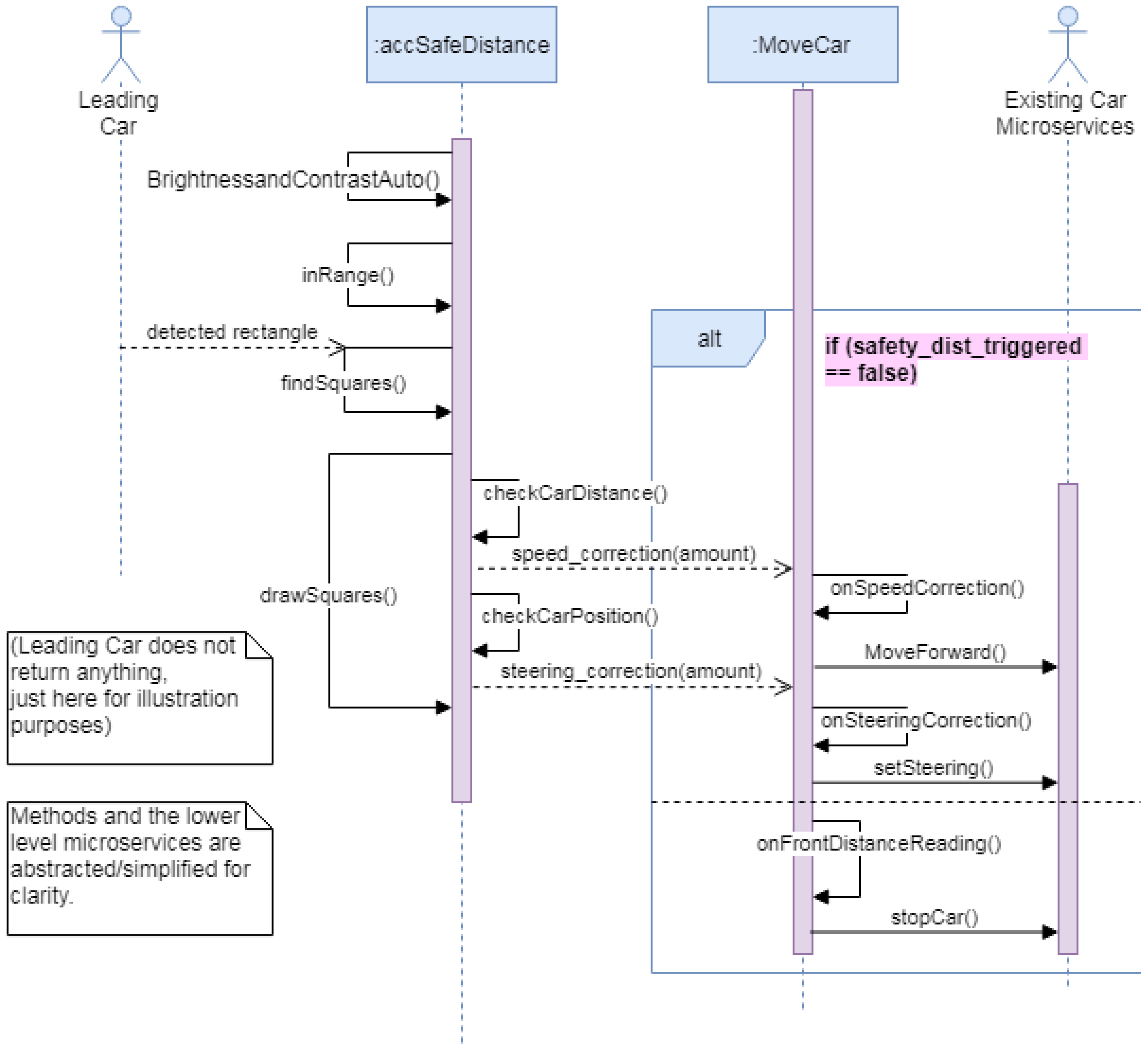

Leading Car
:accSafeDistance
:MoveCar
Existing Car Microservices
BrightnessandContrastAuto()
inRange()
detected rectangle
findSquares()
alt
if (safety_dist_triggered == false)
checkCarDistance()
speed_correction(amount)
onSpeedCorrection()
drawSquares()
checkCarPosition()
MoveForward()
steering_correction(amount)
onSteeringCorrection()
setSteering()
onFrontDistanceReading()
stopCar()
(Leading Car does not return anything, just here for illustration purposes)
Methods and the lower level microservices are abstracted/simplified for clarity.

Fig 1. Sequence Diagram of the Leading Car, Safe Distance Algorithm

The list of the hardware components on the car

- BeagleBone Blue board
- Raspberry Pi 3 mod B board
- Battery (LiPo 7.4V 1200mAh 30C)
- Electronic Speed Control (ESC) unit (SKYRC Cheetah 60A Brushless ESC)
- Steering servo (Hitec Midi-Servo HS-5245MG)
- Motor (SKYRC Cheetah Brushless ESC)
- Ultrasonic sensors (front and rear) (Devantech SRF08)
- Infrared sensors (left and right) (Sharp GP2Y0A41SK0F)
- Raspberry Pi Camera Module v2

Intersection Definition

- Length of a lane
  - 3 lengths of a car long, plus the safety distance between cars (our car is the third one, so with the length of two cars in front).
  - Safety distance between each car = half a car length
- Width of a lane
  - One length of a car for each lane.
- The only lane that will have a stop sign will be ours.
  - Stop sign height: 15 cm
  - Stop sign shape = octagon
  - Stop sign color = red
- Our lane will have a (possibly) separate sign placed below the stop sign preventing our car from going to a certain direction.
- Each lane has a stop line
  - 5 cm thick. Thick enough for the car to be able to recognize it.
- Walls around intersection:
  - There will be walls surrounding the intersection to be able to decrease the background detection of other colors and for decreasing the shadowing on the floor from the window reflexion and corridor light.
  - The walls might not be placed if the light conditions are good in the room that we are planning to demo.
- Lighting conditions:
  - The artificial light of the entire room will be turned on.
  - The curtains for the window will be totally closed if it is a sunny day outside, in order to avoid light reflection and shadowing.
- Room:
  - The room chosen for the testing and the demonstration is milla/void.
- Cars on the intersection:
  - There will be a car in the front lane and another one on the lane of out vehicle. They will be there stopped on the stop line before we arrive.

Intersection Scenarios

1. Following Car, handle empty intersection
    a. Cars on the right and front, both stopped on the line already
    b. Leading car starts 1 lengths of a vehicle with its corresponding safe distance in between, counted from the stop line.
    c. Leading car goes up to stop line and stops.
    d. Our car will stop within safety distance from the vehicle in front.
    e. Leading car waits for the other vehicles already present at the intersection to leave it in their corresponding order.
    f. The cars other cars leave the intersection.
    g. Our leading car leaves the intersection in any direction.
    h. Our car goes forward towards the stop line and stops behind it.
    i. Our car waits for instructions and goes in the chosen direction.
    j. Our car leaves the intersection successfully.

2. Following Car, with two vehicles in the intersection
    a. Leading car starts 1 length of a vehicle with its corresponding safe distance in between, counted from the stop line.
    b. Leading car goes up to stop line and stops.
    c. Our car stops behind the leading car within safety distance.
    d. The intersection is empty at the moment, but there are two cars approaching their corresponding stop lines.
    e. Leading car leaves the intersection first.
    f. The car in the front lane arrives a bit before the car the right lane, but both cars arrives before our car reaches the stop line.
    g. Our car has now stopped before the stop line.
    h. Then the car in the front lane, leaves first. Our car waits.
    i. The car on the right lane goes next. Our car waits.
    j. Then the intersection is empty and we are next.
    k. Then our car waits for direction instruction to leave the intersection.
    l. The car leaves the intersection on the chosen direction.