
CHALMERS

# Self-driven Miniature Vehicle for Carolo Cup 2015

Ashfaq Farooqui
ashfaqf@student.chalmers.se

Johan Eriksson
guserijobi@student.gu.se

May 22, 2015

# Contents

# 1 Introduction

Carolo Cup is an international student competition for self-driven miniature vehicles organized annually in Germany since 2008. Inspired by the DARPA Grand Challenge, the competition Carolo Cup provides student teams from universities the opportunity to develop and implement 1:10 scaled autonomous model vehicles with akerman steering, and compete with other teams. The teams compete in different disciplines characterized under static and dynamic events. The challenge is to realize the best possible vehicle control in different scenarios inspired by a realistic environment.

The competition is divided into different disciplines, as follows:

- *Parking:* The car is expected to park along the right side of a straight strip, three different lengths 700, 630 and 550mm are distributed along the right side. Obstacles emulating parked cars can be present in these spaces increasing the challenge. A maximum of 200 points can be scored in this event.

- *Lane-Following:* The car is expected to drive on a scaled track, representing a rural road. The track has lane marking on either side and also dashed lines in the center. The team is allowed 2 minutes to continuously drive on this road. Point are awarded based on the total distance travelled and violations committed. A total of 200 points can be secured in this event. To increase the complexity, the organisers can remove one or more lane markings.

- *Lane-Following with Obstacle Avoidance:* This discipline extends on the previously done lane following. The car is expected to drive for a maximum of 3 minutes along the same road but also be able to avoid stationary and dynamic obstacles along the road. The car must follow driving rules by giving proper indications while overtaking, stopping at an intersection before proceeding etc. A maximum of 250 points can be secured in this discipline.

- *Presentation:* This is the only static discipline at the competition, here the team is expected to present and motivate technical and non-technical choices made for the car. The presentation is then compared with the running car to assign points to the team. A maximum of 350 points can be gained.

This was the third time a team from Chalmers participated at the competition. Though a new team participates each year, technically we have always been evolving and learning from our past mistakes. This year, we represented Chalmers as team MOOSE, and were placed 11th of 17 teams with a total points of 328. The main technical contributions of the authors can broadly be listed as:

- Setting up the proxy component, which handles communication between high- and low-level boards.

- Hardware support for the car during testing and competition

- Involved in integration phase of the car supporting test and integration team
- Setting up the operating system in order to have a stable car

Apart from this, a considerable amount of time and effort was put into team management, communicating between different sub-teams and managing resources.

This report aims to provide the reader with a general system overview of hardware and software. Furthermore, we put highlight bottlenecks that we experienced and consider important for next years team to consider and improve upon.

# 2 System Architecture

The overall system is designed in a modular, layered structure. Figure 2 shows the hardware structure of the complete system, pin values and corresponding signals are explained in table 1, and figure 1 shows that of software. The components can be divided into two layers, the platform independent model(PIM) and platform dependent model(PDM). The platform independent layer runs on the odroid and consists of high level logical functions, namely Proxy, Driver, Lane-following etc. All of which run an Open Source Development Architecture for Virtualization of Networked Cyber-Physical System Infrastructures(OpenDaVINCI).

An overview of the software elements are shown in figure 1, the different components communicate over an UDP stream in the form of custom defined messages. OpenDaVINCI is designed to improve development by providing a simulation environment, that does not require any code change when running the algorithm either in the simulation or on the actual hardware. This enables testing of algorithm in real-time without the need for always having the hardware present. Keeping in-line with this idea, the proxy provides an interface to the real world, by replacing proxy data, with prerecorded dummy data, algorithms can be tested and evaluated before running them on physical hardware.

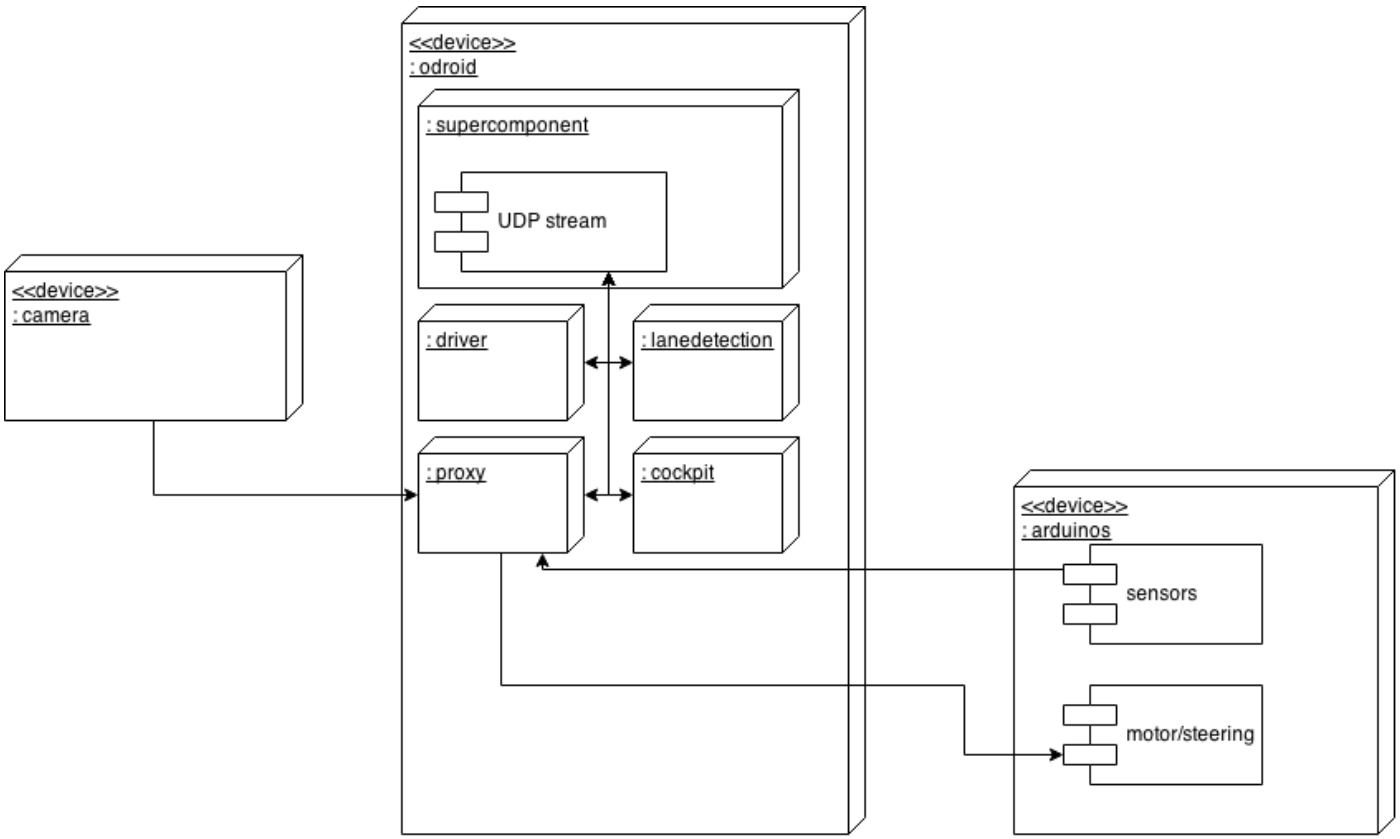


Figure 1: Software overview

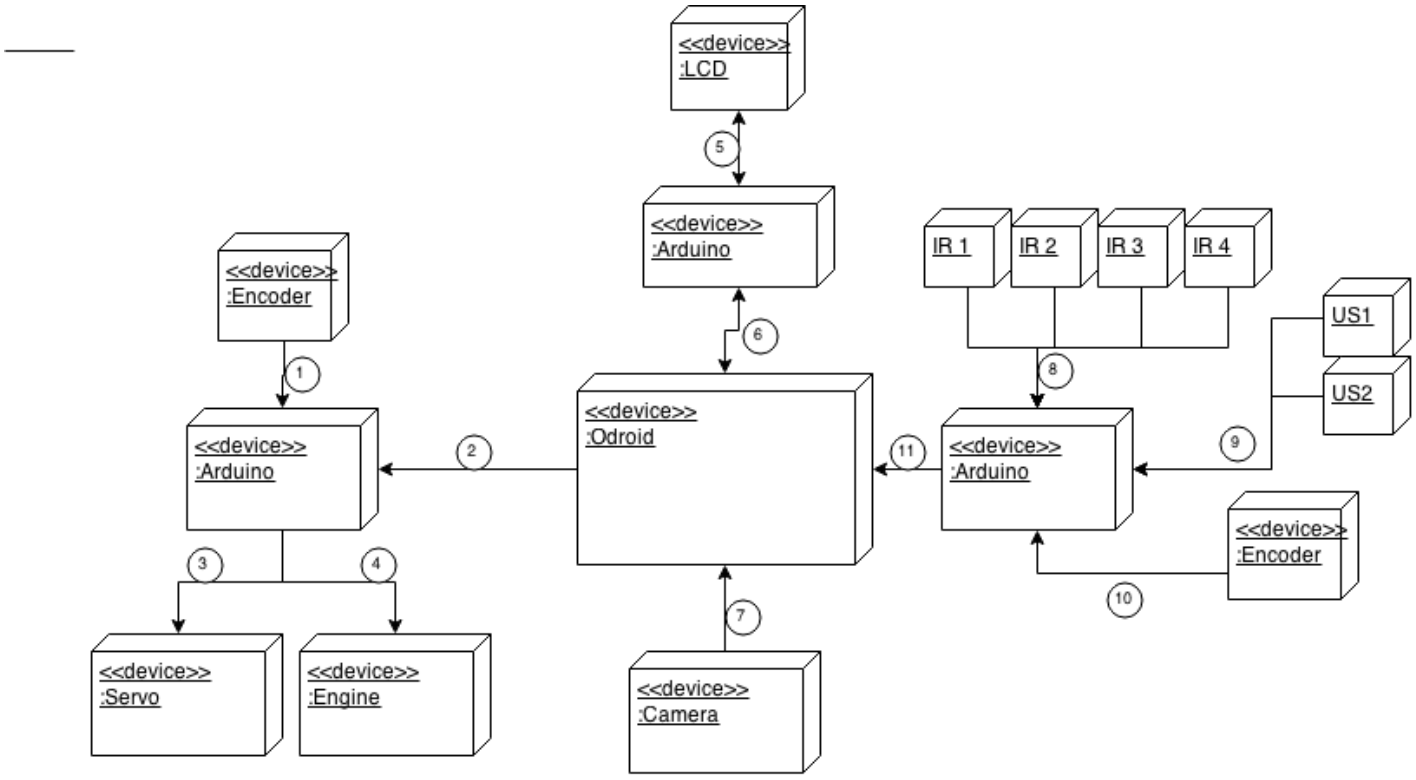


Figure 2: Hardware Layout

The NR in this table corresponds to the connection type and data used in diagram 2.

| NR | Signal |
|---|---|
| 1 | High/Low |
| 2 | Serial data as explained in 4.3 |
| 3 | PWM signal |
| 4 | PWM signal |
| 5 | uses ADAfruit LCD library |
| 6 | Serial data for bash script |
| 7 | Camera data over USB |
| 8 | Analog signals |
| 9 | i2c |
| 10 | High/Low |
| 11 | Serial data as explained in 4.2 |

Table 1:

## 2.1 Hardware Pin connections

Pins used for the actuator arduino communicating with the Motor/Servo: Pins for Motor and Servo:

```
int motorPin = 6;
int steeringPin = 9;
```

Pins for red LED - stop/break lights:

```
int stopLed1 = 24;
int stopLed2 = 29;
int stopLed3 = 23;
```

Pin for blue LED - RC indicator pin:

```
int blueLed = 40;
```

Pins for yellow LED - indicators lights:

```
int rearLeftLed = 22;
int frontLeftLed = 43;
int rearRightLed = 25;
int frontRightLed = 42;
```

Pin that reads wheel encoder data:

```
int wheelEncoderPin = 3 //note this is the interrupt pin on arduino
```

Pins used for sensor arduino that reads data from the sensors:
Pins that reads infrared sensor data:

```
int irLeftSide = 0;
int irLeftBack = 1;
int irRightBack = 2;
int irRightSide = 3;
```

Pin that reads the light-sensor(Lightning conditions) data:

```
int lightSensor = 7;
```

Pin that reads wheel encoder data:

```
#define WHEEL_ENCODER_PIN          0
```

Ultrasonic sensors are connected to the I2C bus which is on pin 20 and 21 for Arduino Mega in our case. Here we use Wire library. The addresses used in our case are

```
#define ADDRESSBACK 115
#define ADDRESSFRONT 112
```

# 3 Dataflows and Datatypes

OpenDaVINCI is designed to operate as a distributed system. The backbone of any distributed system lies in the communication of the different modules within the system. In our case, all communication is handled through the UDP stream. All data that is transferred at the UDP stream can be listed as:

- Sensor Data
- Vehicle Control Data
- Shared Image Data
- Lane detection Data

## 3.1 Data Types

### 3.1.1 Sensor Data

This structure consists of all sensor data in the form of a associative array. The snippet of the .odvd file used to generate this data structure is as shown:

```
message SensorBoardData {
    map<uint32, double> distances;
}
```

All sensor values are maintained in this associative array, physical mapping of sensor and its position in the associative array are currently hard-coded and maintained in the configuration file.

### 3.1.2 Vehicle Control Data

This structure contains data required to control the motion of the vehicle. This includes *speed*, *steering angle*,*brake leds* and *indicator leds*. The vehicle control data originates from the driver component, the proxy component takes this data and performs actuation.

```
message VehicleControl {
    int speed;
    int steeringAngle;
    bool brakeLeds;
    bool leftIndicator;
    bool rightIndicator;
}
```

### 3.1.3 LaneDetection Data

The lanefollowing component is responsible to read image data coming from the proxy, process the image to find lane markings and calculate the goal line using the vanishing point algorithm. The processed data, needs to be sent to the driver component which makes driving decisions. The data structure used in 2015 Carolo Cup can be defined as follows: (The following snippet is just to give an idea, and was not used directly on the system)

```
message LaneDetectorDataToDriver {
    CustomLine leftGoalLines0;
    CustomLine rightGoalLines0;
    CustomLine currentLine;
    RoadState roadState;
    int confidenceLevel;
    int confidenceLevel_goalLine;
    bool noTrajectory;
}

message Lines {
   enum RoadState{
        NOT_SET = 0,
        NORMAL = 1,
        INTERSECTION = 2,
    };
    Vec4i leftLine;
    Vec4i rightLine;
    Vec4i dashedLine;
    CustomLine goalLine;
    CustomLine goalLineLeft;
```

```
    CustomLine currentLine;
}
message LaneDetectorData {
    uint32 m_frame_count;
    Lines m_lines;
    string m_classification; // Inspection classification
    LaneDetectorDataToDriver m_dataToDriver;
}

message CustomLine {
    Point p1;
    Point p2;
    float slope;
    int32 polygonIndex [default = -1];
}

message Point {
    double x;
    double y;
}
```

The current code works but it does not use odvd generator to generate data structures, this will limit the development of this algorithm further. It is highly recommended to make the lanefollowing component to adopt to using generated data structures before making changes to the algorithm.

### 3.2 Data flow between systems

Apart from the data flow within the UDP stream, there is also interaction between the arduino systems and the odroid. The proxy module is the communicator module in the OpenDaVinci System. This is discussed in greater detail under the Proxy (4) section.

## 4 Proxy

The above sections discuss about the overall architecture. As mentioned previously, to allow for offline testing and evaluation without code changes, the proxy component was created. The main task of the proxy is to interface real-world with the algorithms. In order for this, proxy does the following tasks:

- Interface camera
  - open a stream to the camera

  - read data from the camera and add it to the shared memory space
  - Send image header regarding shared image

- Interface sensors
  - open serial port and configure as input on start-up
  - setup an interrupt routine to run a function on reception of valid data
  - on reception of data validate and send across the UDP stream

- Interface actuators
  - open serial port and configure as output on start up
  - on reception of new vehicle control data, forward it to the arduino

## 4.1 Interface Camera

The proxy supports a regular webcam supported by openCV and uEYE camera. By specifying a value in the configuration file, either type of camera can be used.

```
proxy.camera.name = WebCam
proxy.camera.type = UEYE # OpenCV or UEYE
proxy.camera.id = 0 # Select here the proper ID for OpenCV
proxy.camera.width = 752
proxy.camera.height = 480
proxy.camera.bpp = 1 #3- openCV, 1-UEYE
```

The above snippet shows the camera settings in the configuration file. These values are used to create the image headers. bpp defines the number of channels in the image, the uEYE drivers provided in the proxy only support grey scale. Hence, this value must always be 1 for uEYE cameras.

In cases where the proxy was killed or stopped in un-natural ways, the next instance of proxy would open the camera but no image would be available. This error occurred quite often and the only solution found so far was to restart the uEYE camera daemon. It is still unclear if this error was due to the proxy, or the new uEYE drivers for hard float.

Another common error faced was:

```
semaphore error no. 13
```

This would occur either since different users try to access the shared memory segments or when another component, such as cockpit or lane-follower are holding the lock. A good idea for the next team would be to have one person responsible to manage all permissions and setup the basic linux distribution and make it standard, in case of multiple re-installs.

## 4.2 Interface Sensors

Proxy is responsible to collect all sensor data from arduino to allow other components to use it. In the current implementation all data is transmitted as one long string. The following format is used

"i<ir1>,<ir2>,<ir3>,<ir4>:u<ur1>,<ur2>:l<l1>:w<we1>."

here, *i,u,l,w* refer to the type of sensor.

- i-infrared sensor
- u-ultrasonic sensor
- w-wheel encoder
- l-light sensor

Proxy runs a serial port listener, which receives all data that then performs a check to see if the data received is valid. In our case, this is done using *hasCompleteData* function in *serialProtocol.cpp*

```
m_partialData.seekg(0, ios_base::end);
const uint32_t streamSize = m_partialData.tellg();
m_partialData.seekg(0, ios_base::beg);
if(streamSize>70){
  return true;
}
return false;
```

The above snippet shows how data is termed valid. This is a very crude way to perform this task, the authors suggest using netstring or a standardized way of encoding to have more robust communication.

When valid data is available, a callback function *nextString* is called in the proxy, which appropriately splits the string into required parts and converts the string sensor values into integer/float. This method works fairly well here, but at higher frequencies there were noticeable lags in data received. The issue was solved by sampling data from the proxy by clearing the input buffer after each read, but this brings in data loss. The authors suggest using a separate string for each sensor value, which in turn trigger a different call back function, specific for the sensor. This also aligns with the suggestion to use a more robust programming board compared to the arduino.

## 4.3 Interface Actuators

Proxy is also responsible for communication to low level actuator board. The communication is serial and the protocol is text. Each command has a prefix letter after that a value and

a terminating character to ensure completeness of the message. Example commands are as follows:

- s10/ - steering turn 10 degrees right (s - steering, 10 - value, / - terminating char- acter)
- ff5/ - frequency forward 0.5 m/s
- fr5/ - frequency backward 0.5 m/s
- ir/ - turn on indicators right
- il/ - turn on indicators left
- ia/ - turn on indicators all
- is/ - turn off indicators all
- m3/ - Sets the speed to a value predefined in an array with the 3rd value. 0 is used to stop the car.

These messages are sent only when a new value is received from the driver, in order to reduce spamming the low level boards.

# 5 Obstacle Detection

Overtaking discipline is the most challenging dynamic discipline on the competition. The lane following should be working good, an algorithm for overtaking static and dynamic obstacles is needed and an intersection handling as well. Due to limited resources, MOOSE team was unable to test the algorithm, hence the car did not participate in this competition. The idea behind the algorithm is as follows.

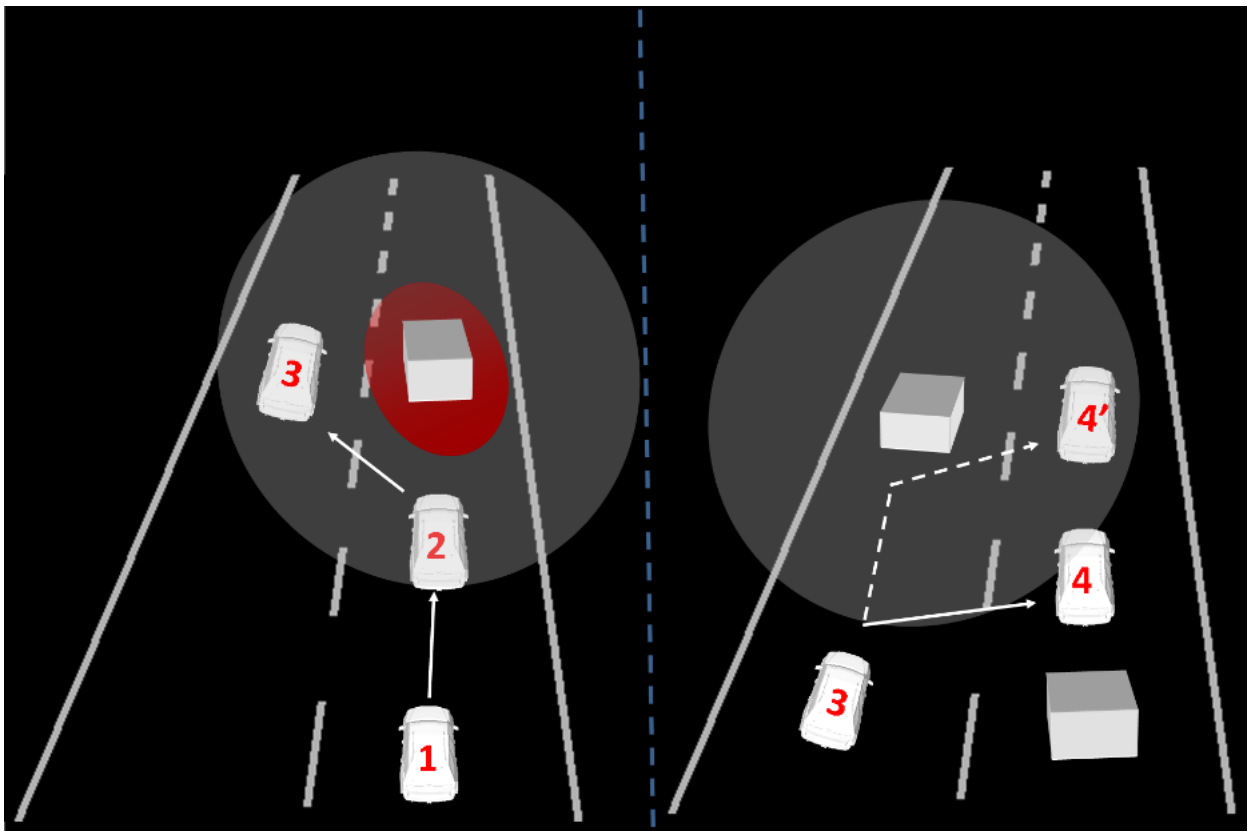


Figure 3: Overtaking concept

1. *Lane-following:* In this state the car follows the lane based on the lane-following algorithm. It is also continuously scanning for any obstacles using the ultrasonic sensor in the front. Due to low response times the car needs to run at a lower speed of about 0.3-0.5 m/s for the current hardware setup.

2. *Obstacle detected:* In this state the car monitors for an appropriate position when to perform an overtake maneuver. A switch lane is performed using 'Time to collision' strategy, which is calculated based on the current speed to determine how much the car can travel before a collision. When the obstacle is about 40cm away, the car gets into the next state.

3. *Switch lane:* The lane-following algorithm always gives two goal lines, one of which is the right goal line. The controller switches from following the left goal line to the right goal line. performing a lane change.

4. *Obstacle overtaken:* Once the car is on the other lane it monitors the side sensors to know when is the obstacle is present. if there is no obstacle detected on both the side infrared sensors then, a command is given to switch back to the left lane, and the car gets back into state one.

# 6 Limitations of the system

This section will focus on limitations that were faced by MOOSE team based on the architecture-software and hardware.

## 6.1 Odroid limitations

The biggest limiting factor was the hardware specifications of Odroid. The processing power was not sufficient. Figure 4 show the plot of processor usage by the different components. As can be noticed proxy and lane-detector use maximum processing power. As this was a limitation, many features like kalman filters and higher predictive algorithms worked with a noticeable lag and could not be tested.

One idea could be to transport the image over the UDP stream, which will reduce reading from memory, the system and be run in distributed mode with proxy and lane-detector running on two different processors. Another idea would be to evaluate how an Ethernet camera could be used for this purpose.

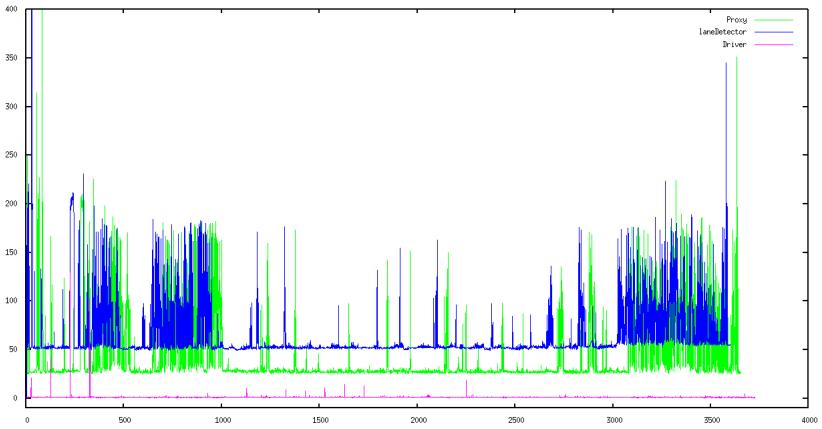


Figure 4: Processor usage

## 6.2 Arduino Limitations

In the current configuration we have two arduino boards controlling the low level electronics. The reason for this was to reduce the time delays introduced when the Atmega IC in the arduino switches between send and receive buffer in the serial port. The disadvantage of this configuration however is, we compromise the behavioral architecture of the system. A number of features such as collision avoidance can be coded on the low level and the system would be more in-line with behavioral architecture keeping in mind PSI and PSM models.

## 6.3 Linux Distribution

We tested a number of various Linux distributions for our system in order to identify and solve issues. The issue was never the distro versions, rather the kernel versions they were built upon. As always with Linux, manage to get unsupported drivers working with the kernel versions is

a huge risk. In our case, the issues affected the camera. It did drop frames, having problem with not locking the USB ports. Solving this bottleneck can not be done by hoping for a "better" version, but by someone who have sufficient Linux skills and can find ways to solve the problems.

### 6.4 Functionality that would improve performance

A good to have feature would be a common logger for all the systems. As OpenDaVINCI is made to work as a distributed system, having a logger that is capable of logging from all components would reduce debugging time. Loggers were built for each module, but they were not synced and hence did not allow debugging between modules.

Another good to have feature would be scheduling components based on user requirements. This was very specific to optimizing performance of the system by running the proxy immediately after driver has finished execution.

We did notice that the driver component on an average, took about 10ms perform calculations and send the vehicle control data. Hence, running the proxy and driver components at the same frequency, but with an offset delay of 10ms in order to reach an optimal performance. However, we have not managed to create this delay between the components but if done, it will significantly improve system performance.

## 7 Good to use practices

Carolo cup is a short duration project at Chalmers, the project begins in early September and ends with the competition in Germany during the beginning of February. This gives a very tight schedule for the participants and the key to success here lies in having a good plan. This section will define a few good practices that will help improve performance for the competition.

### 7.1 Planning

As stated earlier, the key to success is to have a good plan. Gathering the information available and organise this early in the project is of importance. There is a lot of information and advices from the previous years Carolo teams, it is of importance to look through this early in order to not repeat the same mistakes. The group is built up by students from different programs and level of academic studies, it is of importance to identify each other skills and find ways to communicate early in the project. There is a risk that the work will not take off for quite some time in case this does not happen.

### 7.2 Roles

Define the roles early in the project is another key to success. People in the project do have different background and different skills, however, without organisation of the people, no work will be done. There have to be leader(s) assigned that can organise meetings and tasks, the

work of this person does not have to be carried out alone, but the tasks have to be assigned to a person that is responsible for the task to be completed. Trust and responsibility are keywords. Even if tasks are assigned, the leader(s) have to follow up that the work is conducted properly.

We found that dividing the large group into smaller groups responsible for one of the main areas (Lane detection, Hardware, Parking, Overtaking, Test and Integration) would greatly improve our ways of working. For example:

Lane detection - The group is assigned to know every detail about the LD Hardware - The group responsible for building the car and maintain it. Parking - The group responsible for the parking algorithm Overtaking - The group responsible for the Overtaking. Best use to be a part of the LD since it's demanding more of the LD. Test & Integration - The group responsible for testing and integrating the parts.

As a note, no task should have only one member assigned to it. At least two on each task. In case something happen and only one member know how a single function work, it may cause major problems for the team.

## 7.3 Competition

At the actual competition, only a small number of teams will be allowed at the track at the same time. Teams are given a set number of time slots per day with no room of changes. For the team, this require good organisation and planning in order to make most use of the time slot. There are no room to waste time at testing different variable changes, in order to make a single feature perfect, the time have to be dedicated to a set number of tasks, if one is done tested, move on to next task.

Data can be analysed off track, with use of the correct tools in order to gather data, (video footage, sensor logs, more logs). Everything (both software and hardware), should be prepared and make and work properly before the travel, since there will be many other things to do at spot there.

## 7.4 More work with hardware

A mistake that the 2015 Carolo Cup group did was to develop mainly in the simulation environment. This environment is very helpful to run basic tests of the algorithm, but only when the algorithm is run on the actual hardware can we be sure of its working. Videos are simulating the track quite well, but since the car does not respond to the movement suggestions, there is no way of actually knowing if the car would had responded correctly in the new position it would had taken.

Furthermore, physical problems such as light, shadow and reflections have to be dealt with in reality, but not in simulation. Many member of the team will need to know how to deal with this. The integration phase of the algorithm is complicated and issues that could not been for seen will come to light. It is the period when the team learn the most about the car, but also a period that require dedication and clear goals.

The earlier the integration and work at the hardware can start, the better it is for the team.

# 8 Lessons Learnt

In this section we will list a number of issues that caused problems for us when it comes to non technical things.

Leadership - The lack of clear leadership divides the group. No matter how the group decide to work, it is important to have one or a few leaders that can organise the work. With a few leaders, a responsible for each subgroup could also have the final words in cases of conflicts. Something that would help the project a lot, however, great responsibility is required from all people in the team.

Planning, Organisation & Communication- Hand in hand with leadership. All three parts are very important for a successful project, decide early on how to communicate and stick to the decision. Assign members of the project to roles in planning and organisation, involve each other.

Sharing knowledge - As an example: we had issues with a subgroup that made changes to the Arduino code without telling the other subgroups. It did create unexpected behavior and many hours of error tracking wasted on - nothing. There are several ways to solve this and should be dealt with in the beginning of the project, make people responsible for code versions, use only one version of the code, create check lists for tests. Set rules for testing, always include a member of the test team, etc.

In general, the main issues for 2015 years Carolo was not the technical part, but the organisational issues. We solved all issues, but it took to long to do so, which lead to less time doing relevant work. As for next years students of Carolo, we recommend you to deal with these issues early on, have many meetings in the beginning, take the chance to get to know each other and decide on how you would like to organise the project.

## 8.1 Summary of Recommendations

Read section 8 and 7 for details about these points, the points are only created in order to help the reader summarise the lessons.

1. Divide a large group into subgroups.
2. Assign roles and responsibilities early on:
    - Leader/Leaders roles.
    - Planning responsible.

- Hardware responsible.
- Presentation responsible
- Overall Software responsible.
- Test responsible.
- Integration responsible.
- Organisational/Planning.

3. Involve each other by dividing responsibilities.
4. Make a plan early.
5. Get to know each other by doing workshops early.
6. Share knowledge - who's good on what?
7. Share knowledge - decide on how to communicate within the group.
8. Start with hardware early.
9. Start integrating early.
10. Test a lot in all phases.
11. Take time and do proper analysis.
12. Use the test tools that exist, create new ones ASAP if you find it necessary.
13. Plan your competition.