Infrastructure for BMW Test Car Display System

Part A – Spring 2018

Project Summary

Developed by

Daher Daher daher928@campus.technion.ac.il

Vivian Lawin vivlawin@campus.technion.ac.il

Supervisor

Boaz Mizrachi

Table of Contents

1 Preface

In order to enhance and improve the driving experience in BMW cars in general (and BMW's Autonomous cars in particular), drivers and car testers must understand and feel not only the condition of the environment and the traffic, but also the condition of the road, e.g. fraction of the road, humidity, temperature and angle.

Feeling and experiencing the interaction between the car and road is important for optimizing car's safety and efficiency both for human drivers and autonomous driving.

Our project aims to visualize the conditions of the road and the real-time data collected by the car's sensors and provide a comfortable and elegant user interface to graphically display the results on the vehicle's dashboard screen display.

2 Introduction

2.1 Is it necessary?

Car testing is very essential, and the more the testing procedure is efficient the more the results are reliable, and time is saved. both advantages are highly desired in the era of new technology, and especially in the emerging field of Artificial Intelligence and autonomous cars, enabling a transformation and technological rise in the mobility industry.

When mentioning AI and autonomous cars, the term “Safety” pops up as a key goal. Car testing – and especially when intended to be driven autonomously – is a necessary process for assuring and guaranteeing the safety of the passengers and the vehicle’s surrounding environment and pedestrians.

2.2 Project Goals

The main goal of this project is to provide car drivers and testers with a friendly and easy-to-use application with the ability to graphically display the real-time data received by the car's sensor.

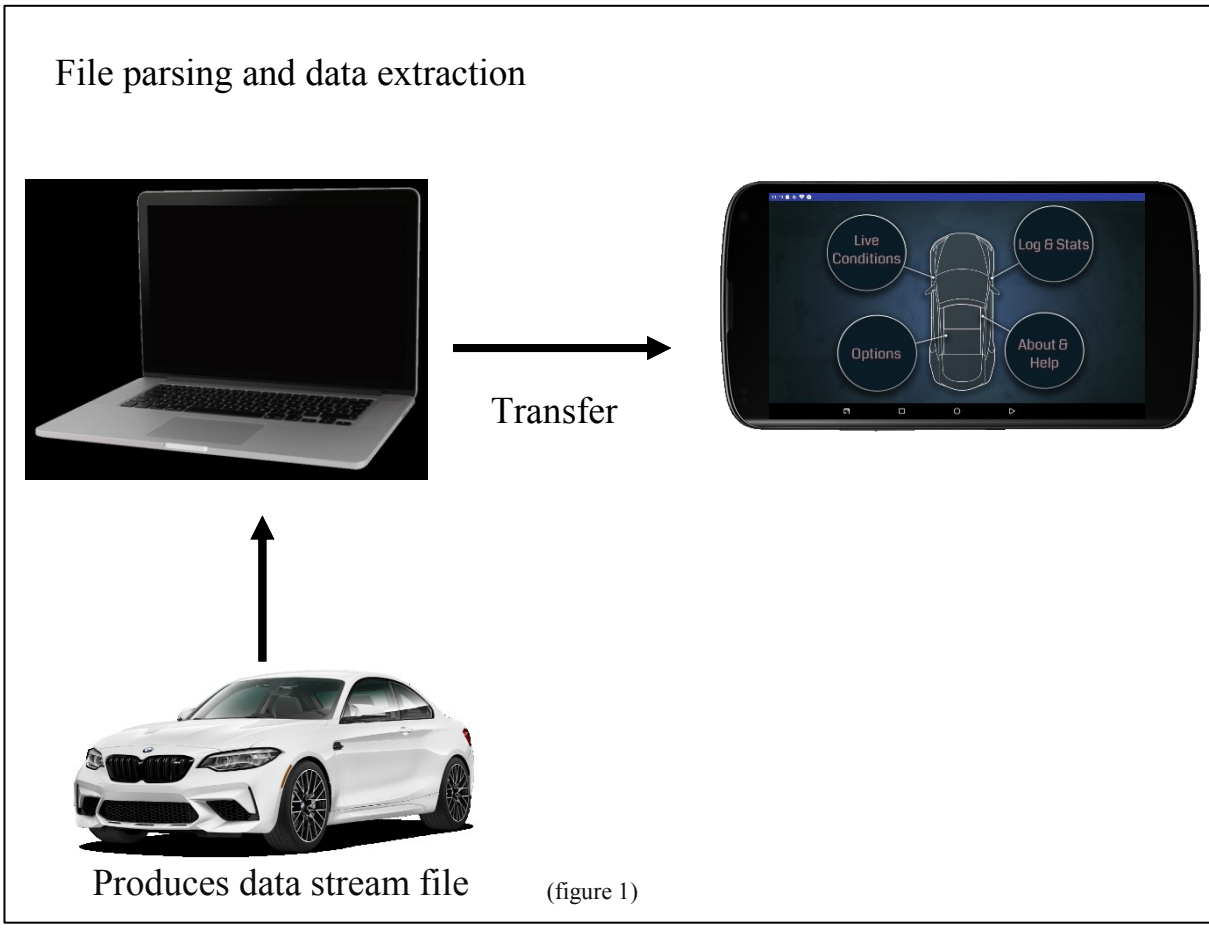
Each of the following features of the application will add to the users experience much convenience, starting with choosing the configurations of each graph, to getting previous results, which are very useful for comparing results and investigating the changes in the results according to changes in the surrounding circumstances.

Full features list:

- Support multiple graphs (Up to 3 graphs at a time).
- Choose different configurations (Resolution, Max value, Color…).
- Support logging and data recording (internal memory file and in UI).
- Time axis scaling.
- Different appearance theme.

3 Concept

The main concept is having two independent parts communicate one another, in order to transfer data (figure 1). The roles are a *Transmitter* and a *Receiver* (client-server). The Transmitter is a laptop which receives file of sensors data streams, parsing it and transmitting the data to the receiver. The receiver is a part of the application installed in the car's dashboard display system. On receiving data, the receiver parses, filters and enqueues the data for further processing by the application (e.g. graph updates, logging, etc.…). Each of the above roles are independent and can be replaces and improved independently.



(figure 1)

4 Technology

The following chapter describes the different technologies used for implementing the project, such like: display screen, data transfer solutions, development environments and developing tools.

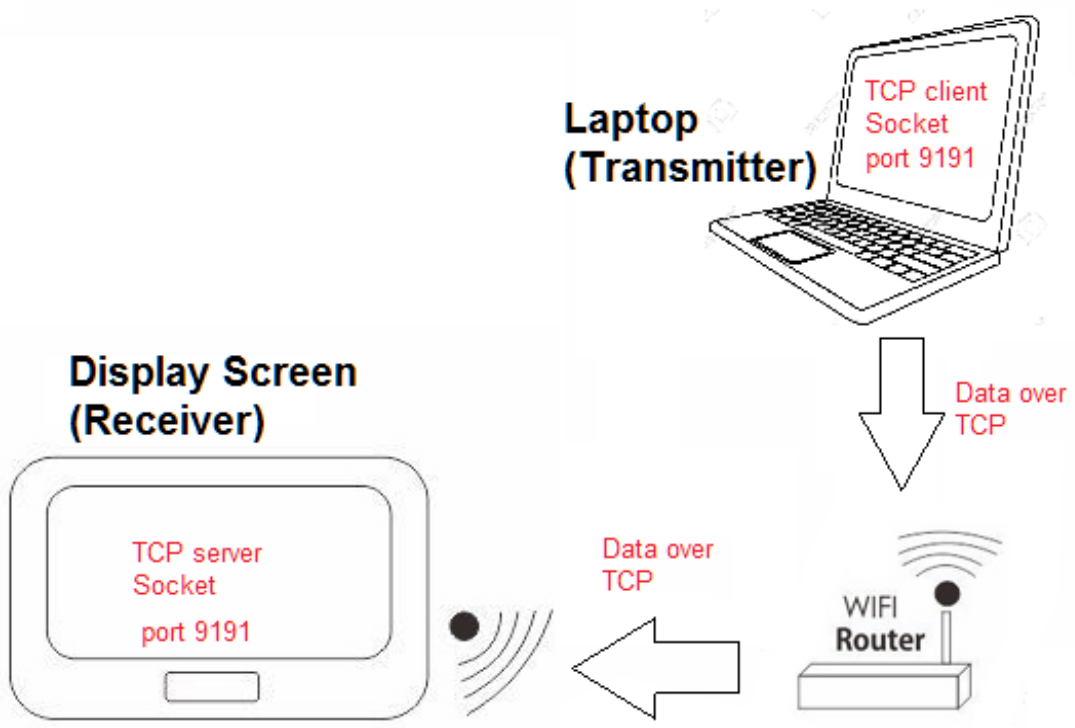
4.1 Display screen

The device that is used for running the application and displaying the real-time results and graphs is GINI Tab V7. Following are the specifications and features of the device:

Native Platform	Android Nougat 7
Release Year	2017
Device RAM	2 GB
Battery Capacity	2800 mAh
Has Removable Battery	Yes
Weight With Battery	280 grams
Max Internal Storage	16 GB
Expansion Slot Type	MicroSD
Expansion Slot Max Size	32 GB
Supported I/O	3.5mm Jack, Micro-USB
Supported Charger Types	Wire
Supported Bearers	Bluetooth, WiFi
Has NFC	true
Screen Type	IPS
Screen Inches Diagonal	7.0 inch
Screen Pixels Width	1280 pixels
Screen Pixels Height	800 pixels
Bits Per Pixel	18
CPU Cores	4
CPU Maximum Frequency	1.3 GHz
Screen Inches Square	22 inch
Screen Inches Width	5.94 inch
Screen Inches Height	3.71 inch
Supported Bluetooth Version	4.0

4.2 Data Transfer Solution

As mentioned in *Chapter 3*, there are two independent parts which aim to transfer data between them (figure 2); the transmitter (laptop), and the receiver (display system application). In this project, we achieve this type of communication using **TCP networking protocol** and **TCP Sockets**. Both the transmitter and receiver must be connected to the same WI-FI network, then open a TCP socket on the same port allowing them to communicate. The receiver acts as a server and listens to incoming connections from the transmitter (client).



(figure 2)

4.3 Software and IDEs

The transmitter is a Java program developed as a Java application, which runs on windows laptop. The IDE used for developing the transmitter is Eclipse, using common Java libraries for TCP networking and sockets.

The receiver is built-in the Android Application for the display system, developed in Android Studio IDE in Java language for android.

- Android Studio IDE 3.3

  - Java over Windows OS

- Eclipse Oxygen 4.7

  - Java over Windows OS

4.4 Graphics and UI components

Most of the graphic components in the Android application are designed using Adobe Illustrator and Adobe Photoshop:

- Adobe Photoshop CC 2015
- Adobe Illustrator CC 2015

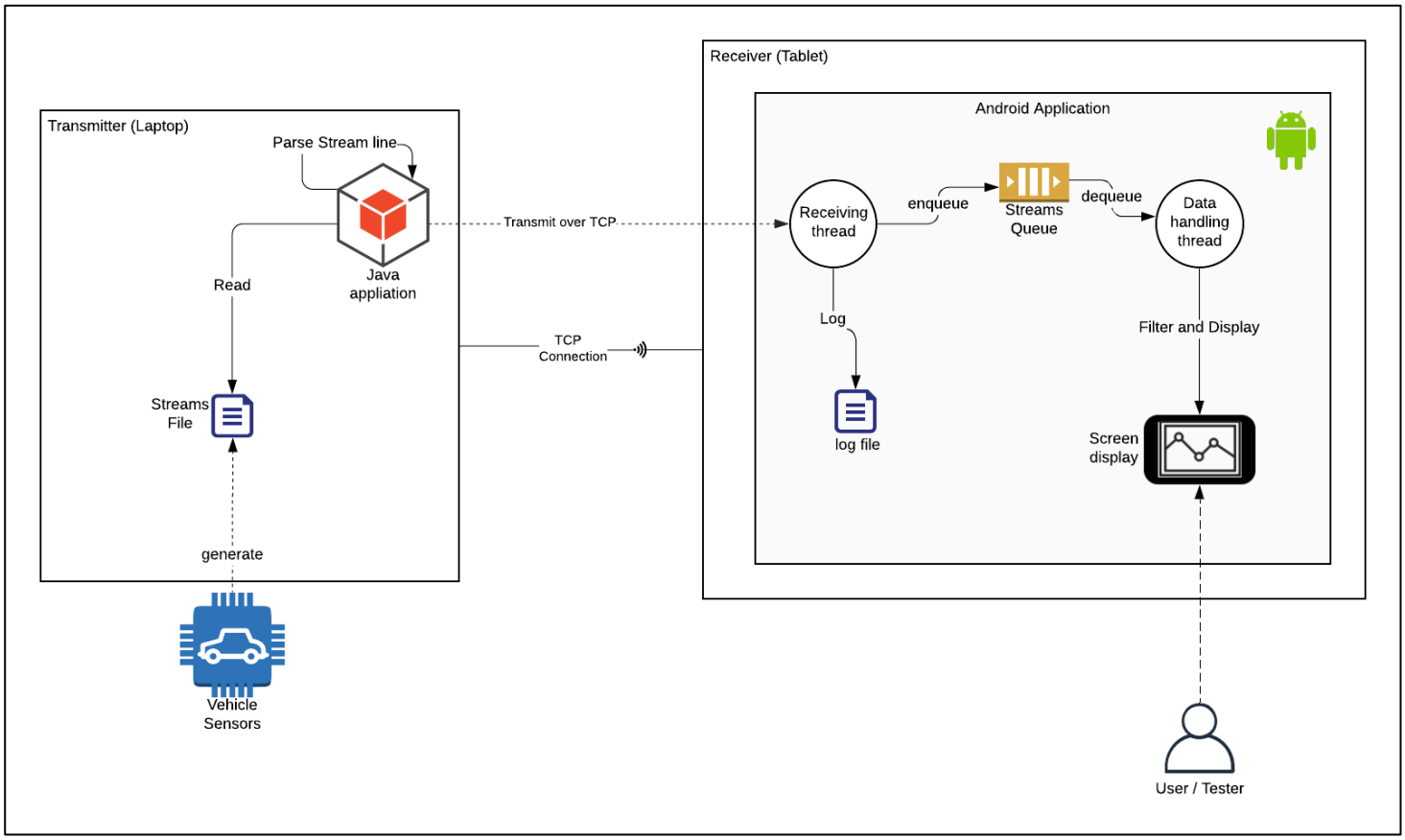
5 High Level Design

As mentioned earlier, the main components are a transmitter, and a receiver.

The transmitter (Java application), reads the streams file which the vehicle system (multiple sensors) produce, and parses each streamline according to the agreed structure. Afterwards, the application transmits data streams over TCP to the receiver application.

The receiver (Android application) contains a process for receiving the data sent over TCP, and enqueues relevant parts of the stream to a Producer-Consumer queue. The receiving thread also logs the data received into a log file located in the tablets file system, allowing the screen to display the loggings of the data. Another process in the receiver is responsible for consuming the data from the queue, filtering relevant data according to the user's desire, and updating the real-time graphs.

(figure 3)

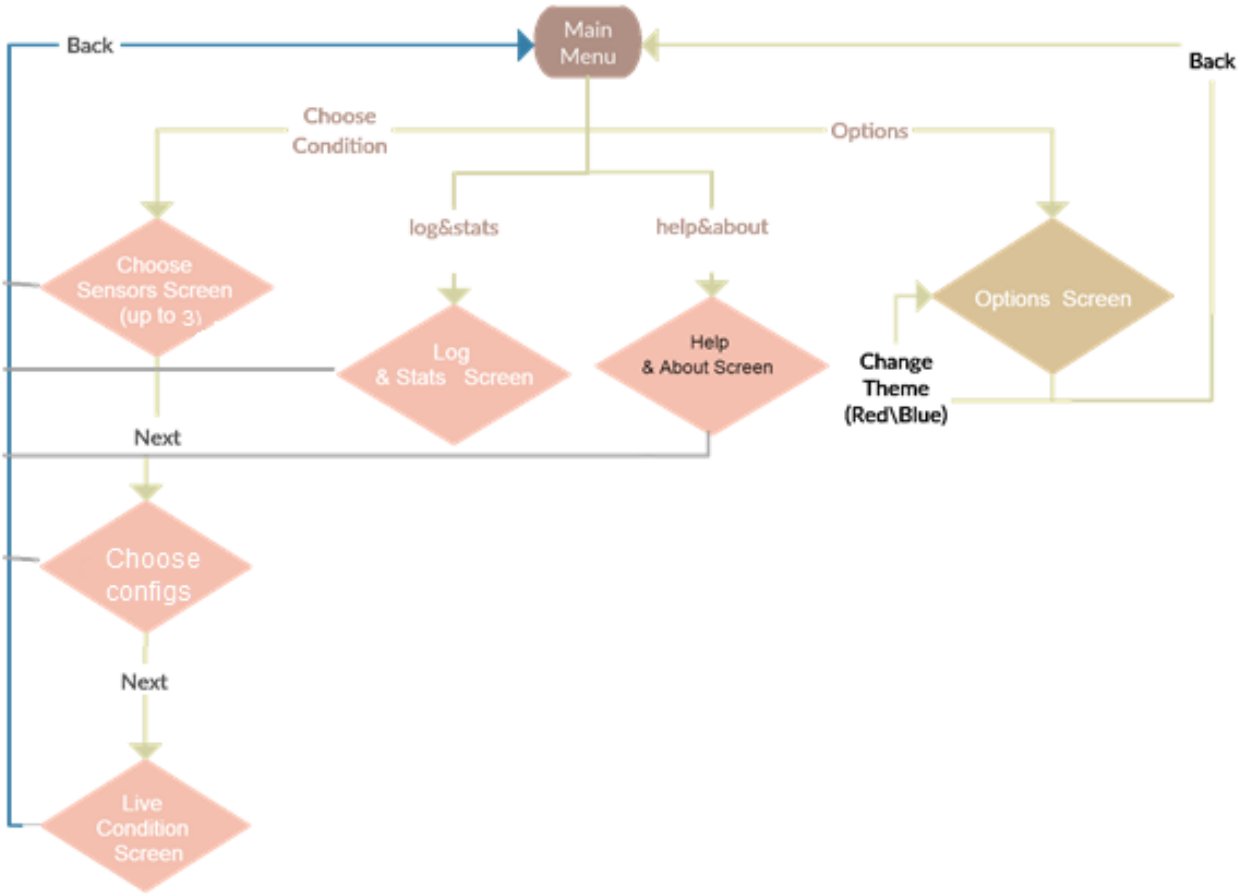


6 Implementation

The following chapter describes the implementation of the project, split into two sections: The User Interface part which describers the interface of the application: flowchart, screens and how to use the application. The second section is the backend part: the low-level implementation of the transmitter-receiver, and the internal processing parts of the application.

6.1 User Interface

6.1.1 Flowchart



(figure 4

6.1.2 Screens and Usage

6.1.2.1 Main page

This screen is the homepage of the application and the first screen that the user sees within entering the application.
In the main page (Figure 1) the user can choose one of the following:

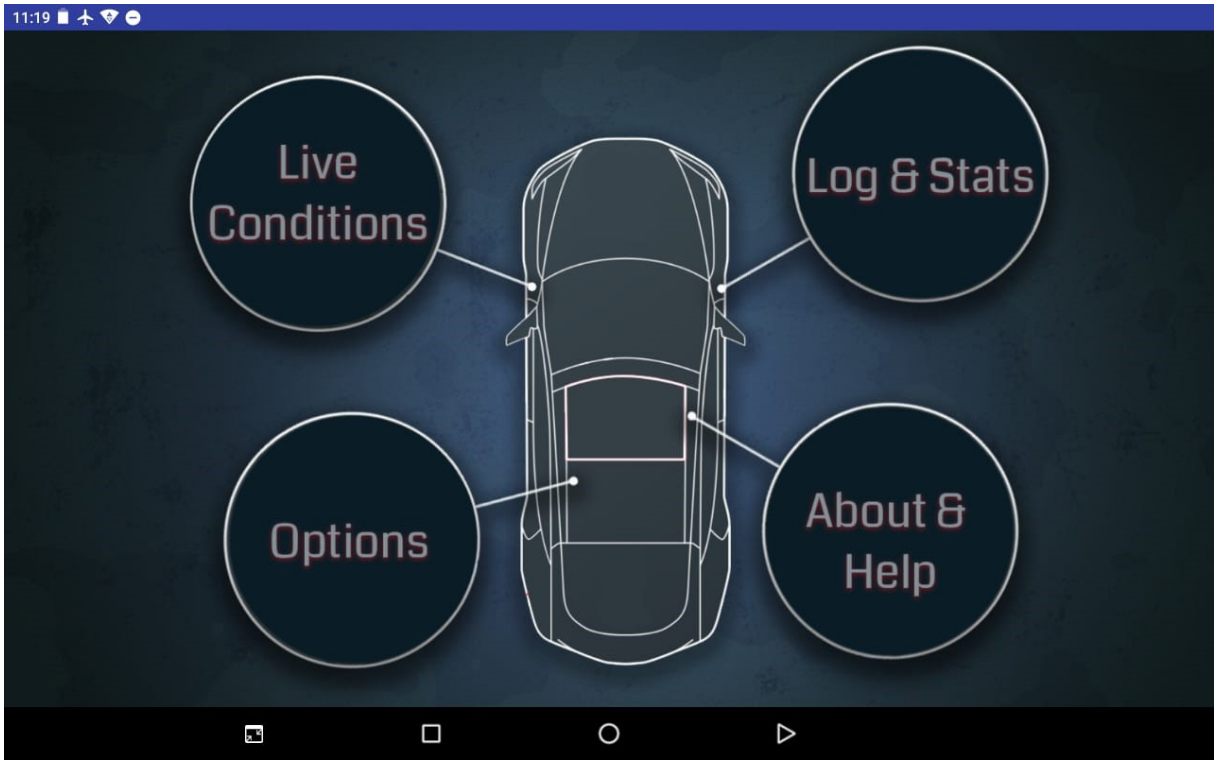


Figure 1
Main Page

User can in addition change the theme of the application (see Options) according to their preferences. This application supports two themes: blue and red).

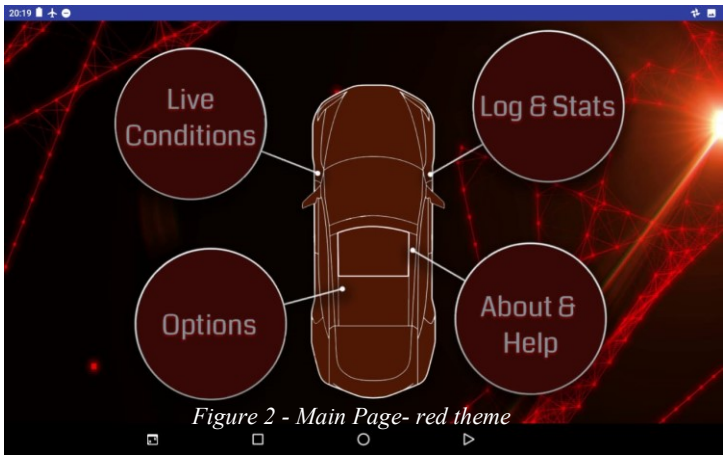


Figure 2 - Main Page- red theme

6.1.2.2 Live Conditions – Choose Sensors

In this page **Error! Reference source not found.**the user will be asked to choose up to 3 of the sensors to be shown in the graphs of real time measurements.

The user can choose to clear all their selections by pressing CLEAR ALL button as seen below. Users can also go back to the previous page or the next one, after they have chosen all the options they desire to be shown, by clicking the appropriate button.

Users cannot proceed to the next screen (using the next button) unless at least one option was chosen.

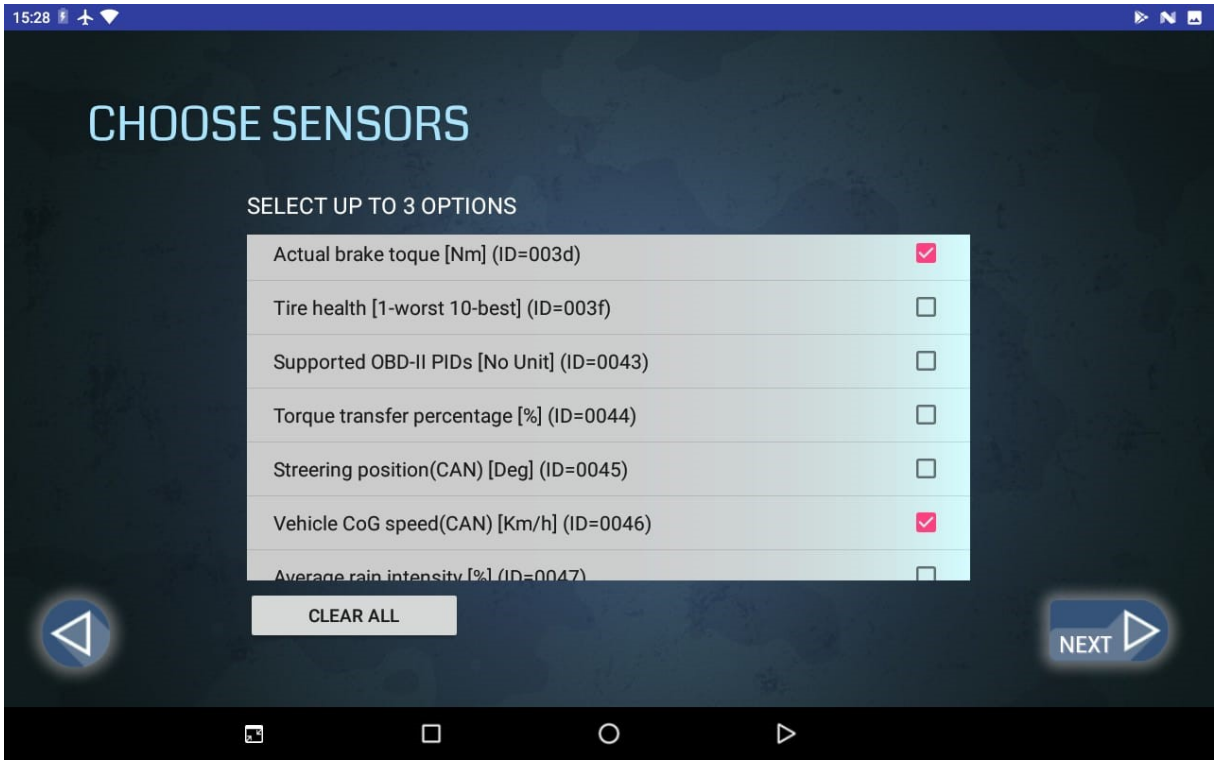


Figure 3

6.1.2.3 Live Conditions – Choose Configurations

After choosing the sensors the user desires to be shown, the next step is to choose the configurations. When clicking the NEXT button, the user is redirected to the next page:

If no specific configurations were chosen, the default configurations (read by sensors configurations file) is used.

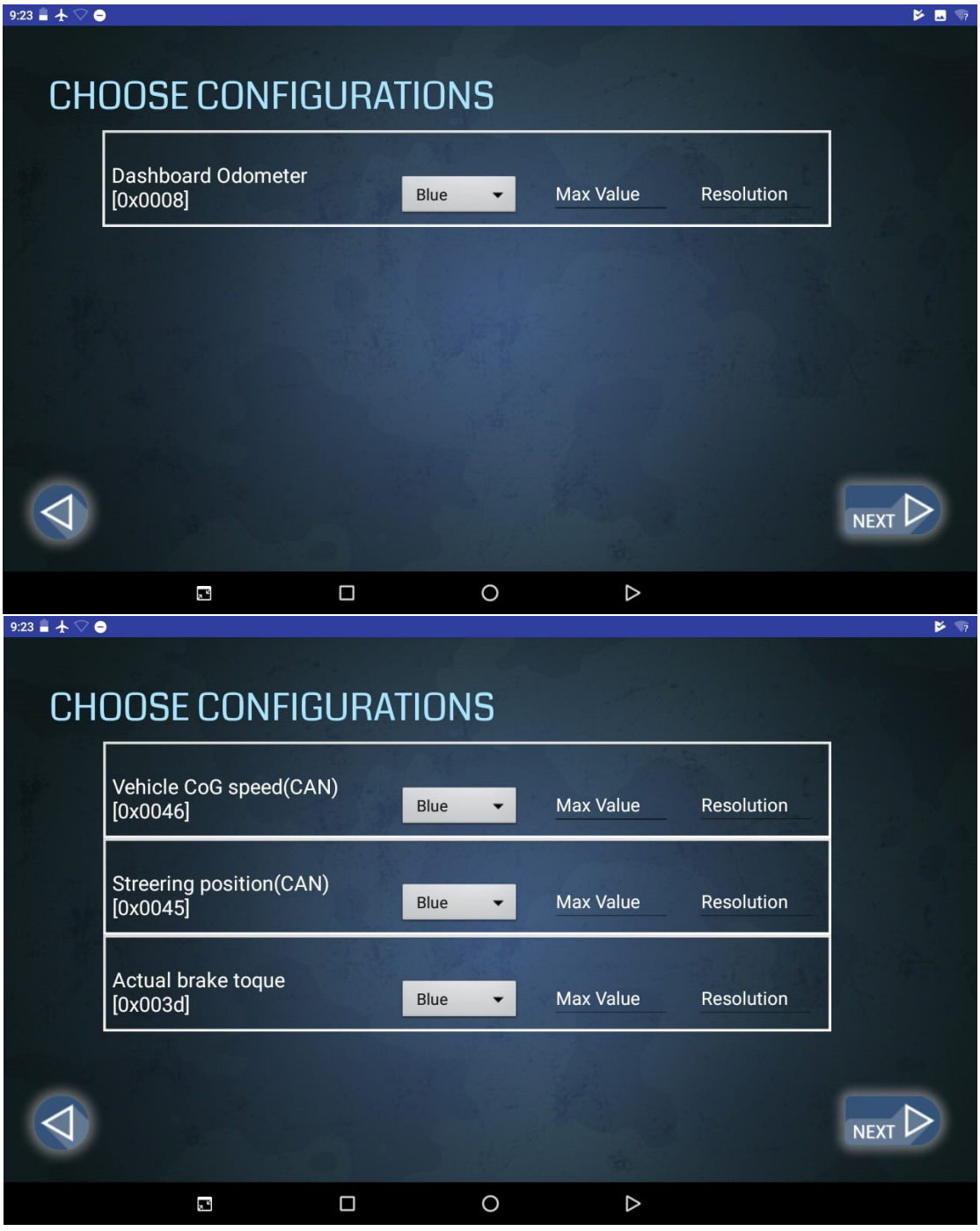


Figure 4, 5

6.1.2.4 Live Conditions

The most significant page – the actual live conditions and graphs, according to the configurations chosen by the user. (Figure 4 and Figure 4 ,*Figure 5*).

On the top right-hand corner, the user can switch the logging option on/off. When the option is On, the log file is displayed in the Log & Stats page.

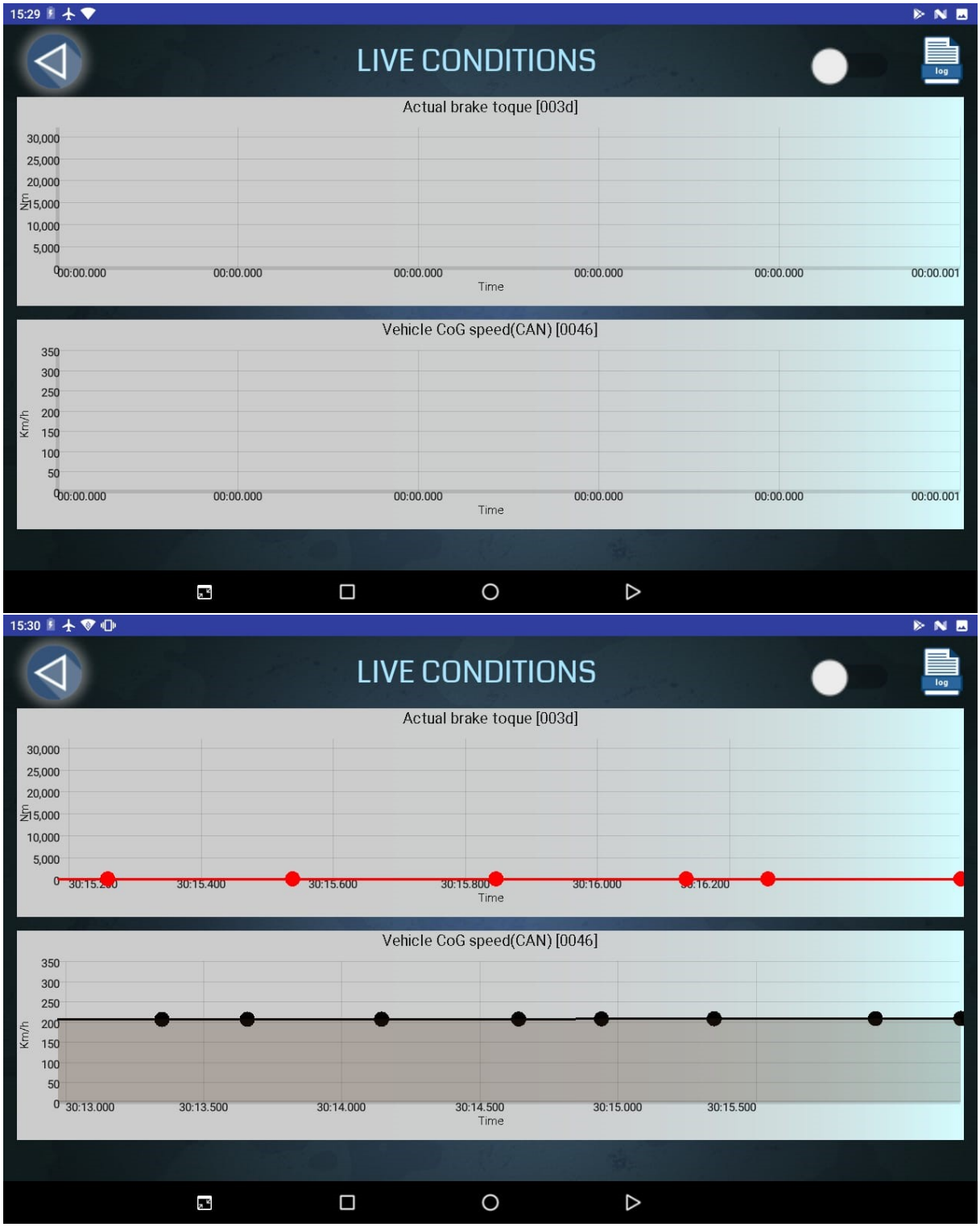
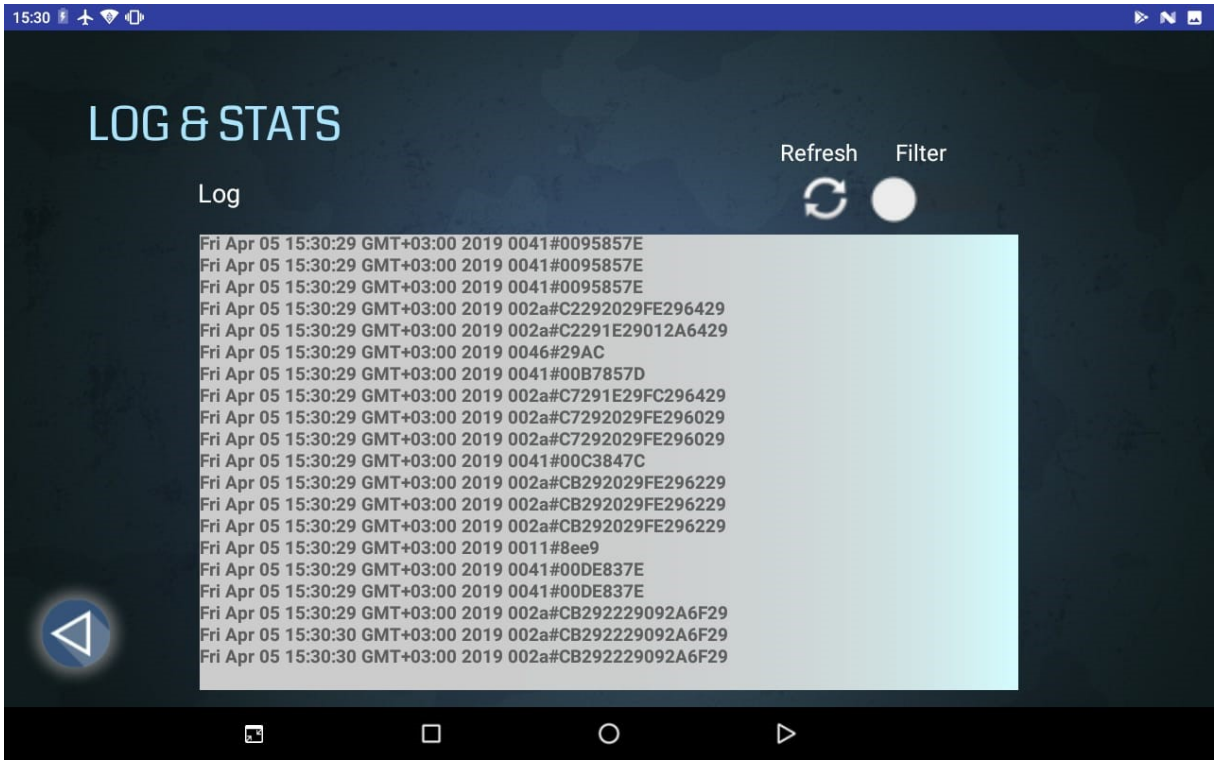


Figure 4 ,Figure 5

6.1.2.5 Log & Stats

In Logs & Stats the user can find log that were recorded. It helps the user compare between results and save previous results which makes this application more efficient. This application provides logs with and without filtering according to the configurations (sensors) chosen in the Choose Sensors. (Figure 6 and 8).



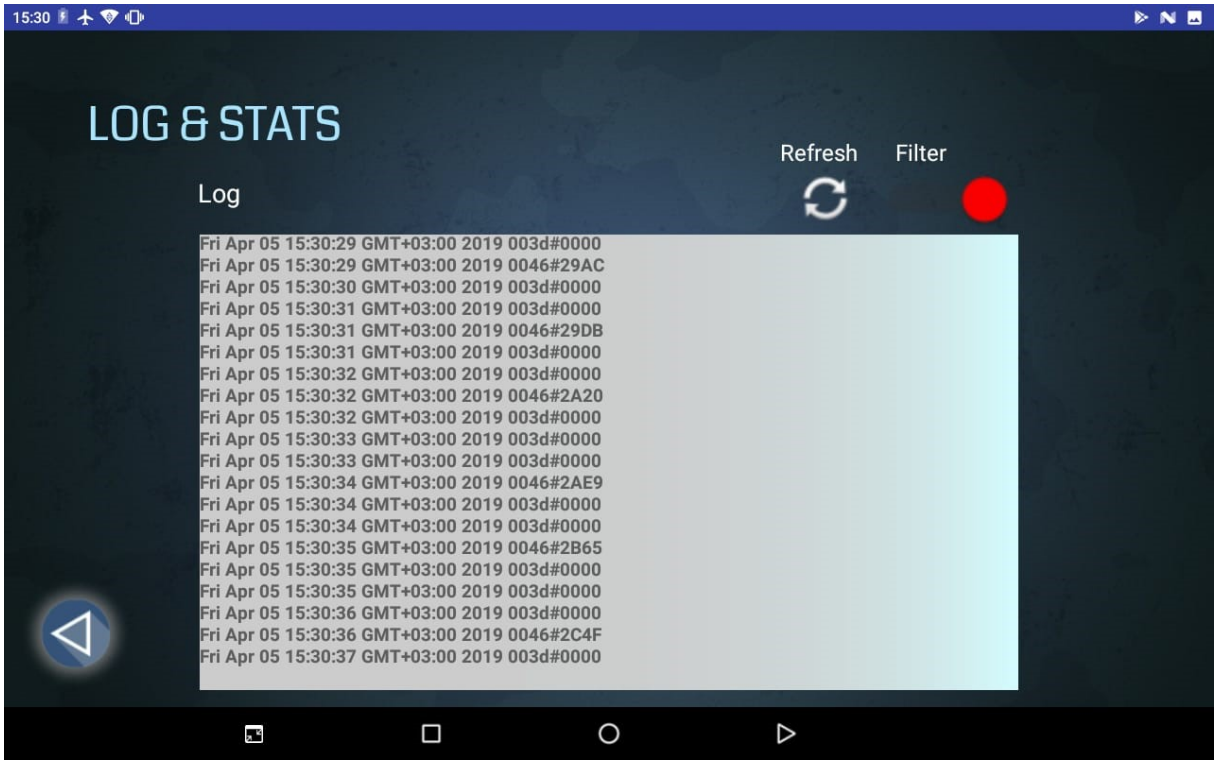


Figure 6, Figure 7

6.1.2.6 Options

In options the user can change the theme of the application (Figure 8 and Figure 9). The application supports two themes: red and blue.

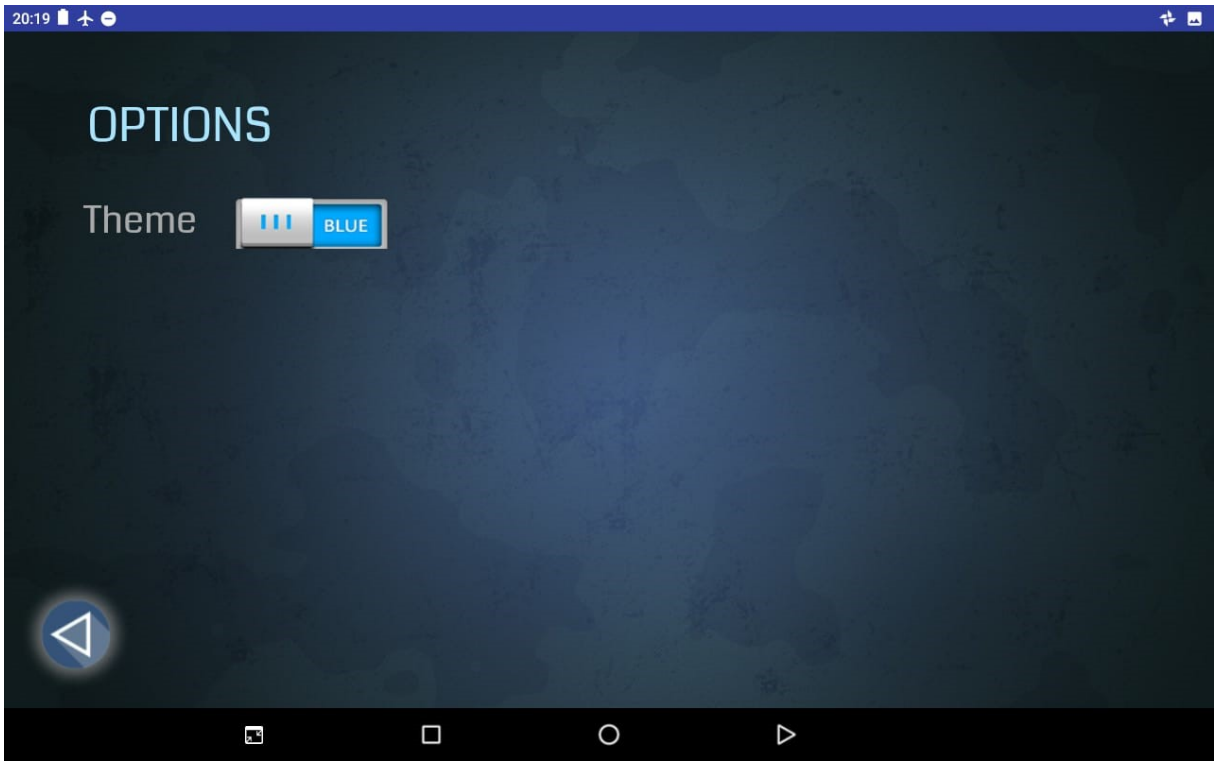


Figure 8

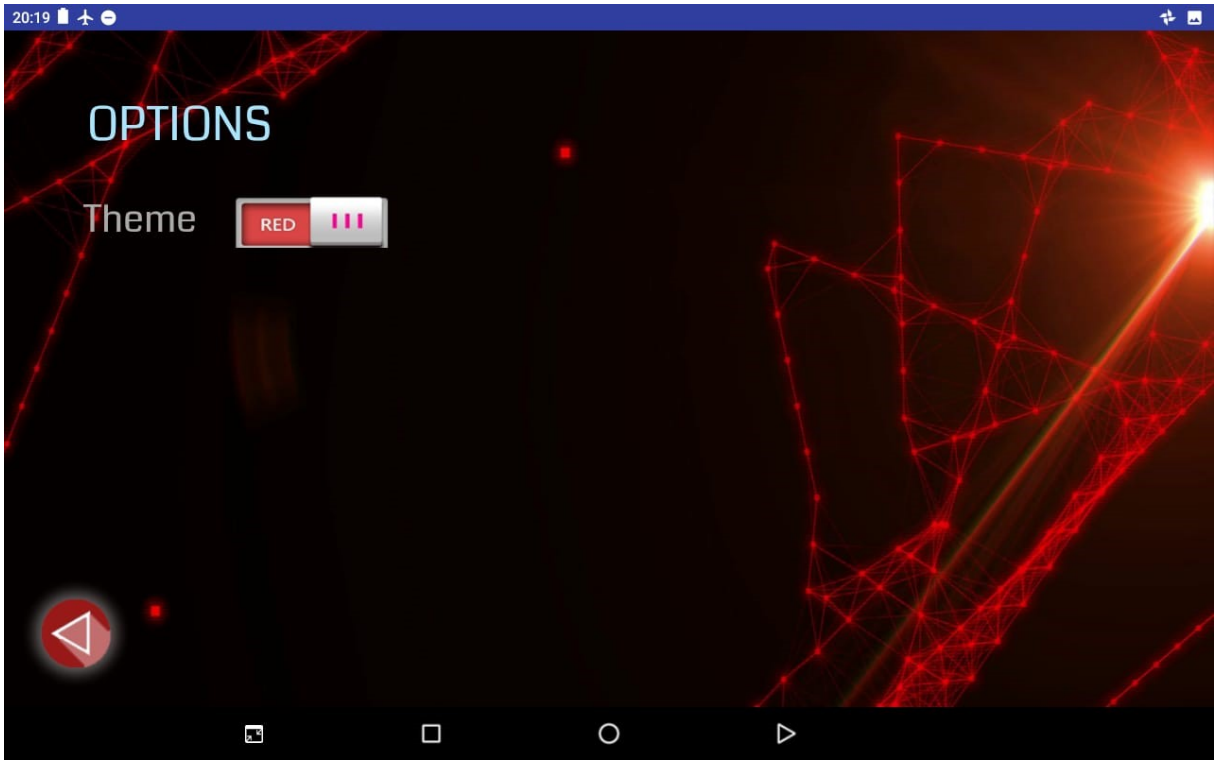
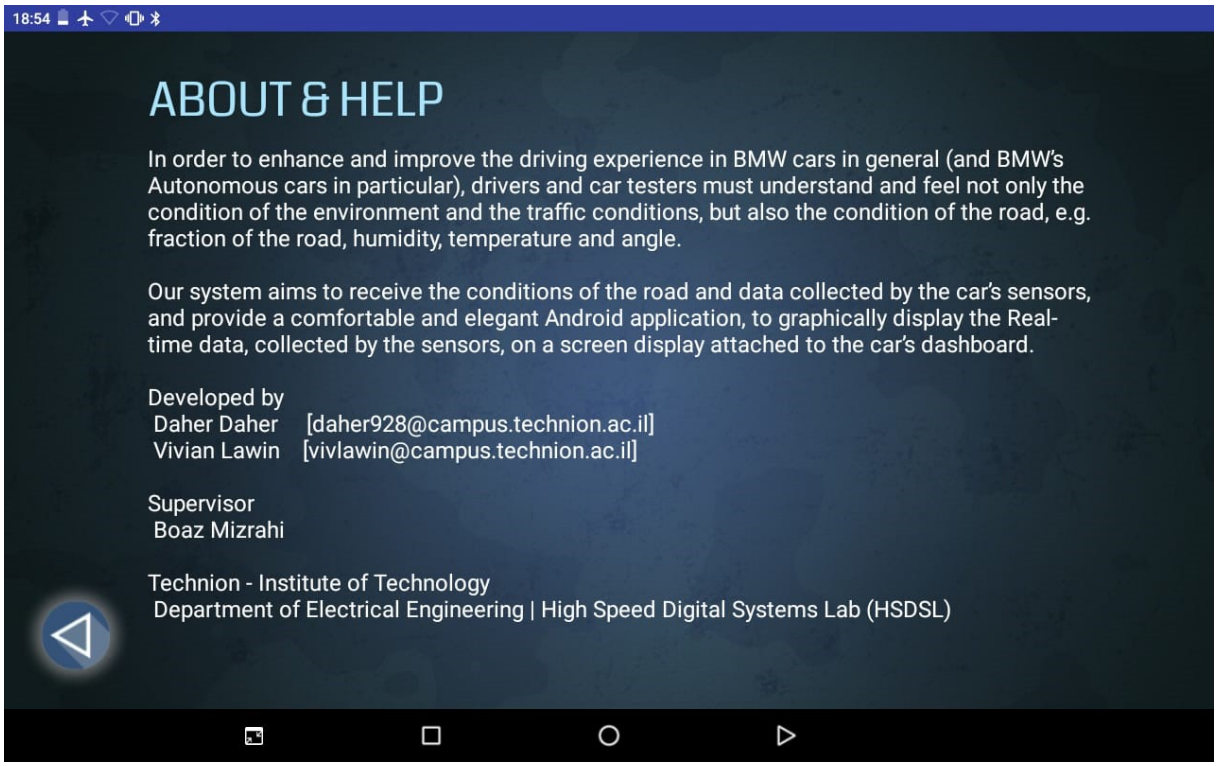


Figure 9

6.1.2.7 About & Help

In this page you can find a brief summary about the application.



6.2 Back-End

In this section, you will find a description of both the transmitter and receiver at a more low-level and parts of the implementation code of the operations they perform, as well as a description of the library used for managing graphs.

6.2.1 Transmitter

The transmitter, as mentioned earlier, is a Java application responsible for collecting the relevant data and transmitting them to the android application for further processing and displaying.

In order to do so, the Java Application sets up a connection over TCP (see chapter 4.2 – Data Transfer Solution)

```
static final String androidId = "192.168.43.156";
static final int port = 9191;
static Socket socket;
static PrintWriter printwriter;
public static void setUpConnection(String Ip, int port) {
        socket = new Socket(Ip, port); // connect to server
        printwriter = new PrintWriter(socket.getOutputStream(), true);
}
```

Afterwards, the application proceeds to read the streams file written by the vehicle's system. The following is an example of the streams file's structure:

```
Start DATA
MWSTRM 000000031001a10000SOH1002f0SOH100120387SOH
MWSTRM 0003ef021001100SOH1001202C4SOH
MWSTRM 0004a20210033011A39D30155236528CB1FFF0100SOH1001100SOH
MWSTRM 0005b4021001202D4SOH1001100SOH
MWSTRM 00075d021001202AESOH1001100SOH
MWSTRM 00088e0210033011A39D30155236528CC1FFF0100SOH10012029FSOH
MWSTRM 0009c8021001100SOH100120291SOH
MWSTRM 000b7a021001100SOH10012027CSOH
MWSTRM 000c750210033011A39D30155236528CC1FFF0100SOH1001100SOH
MWSTRM 000de702100120289SOH1001100SOH
MWSTRM 000f8802100120297SOH1001100SOH
MWSTRM 00106602100033011A39D20155236528CC1FFF0100SOH100120293SOH
MWSTRM 00120b021001100SOH100120297SOH
MWSTRM 00130802900118ec3SOH1004100017A7FSOH
MWSTRM 00131e041002a1027102710271027SOH1003d08CASOH100462710SOH1002a1027102710271027SOH
MWSTRM 00132b021003c7D000000SOH1004100017A7ESOH
MWSTRM 00133e021002a1027102710271027SOH100190100SOH
MWSTRM 00135a031002a1027102710271027SOH1003d08CASOH1004100027A7ESOH
```

(See Appendix for full structure and parsing of the streams file).

For each streamline read, the application parses it into a DataSample object,

```
public class DataSample {
        public int timeStamp;
        public int samplesCount;
        public List<String> sampleIds;
        public List<String> samplesData;
}
```

and sends over the socket every sample data in the DataSample.

```
DataSample ds = parseLine(line);
for(int i = 0; i < ds.samplesCount; i++) {
        String sampleId = ds.sampleIds.get(i);
        String sampleVal = ds.samplesData.get(i);
        send(sampleId + "#" + sampleVal);
}
```

The send method:

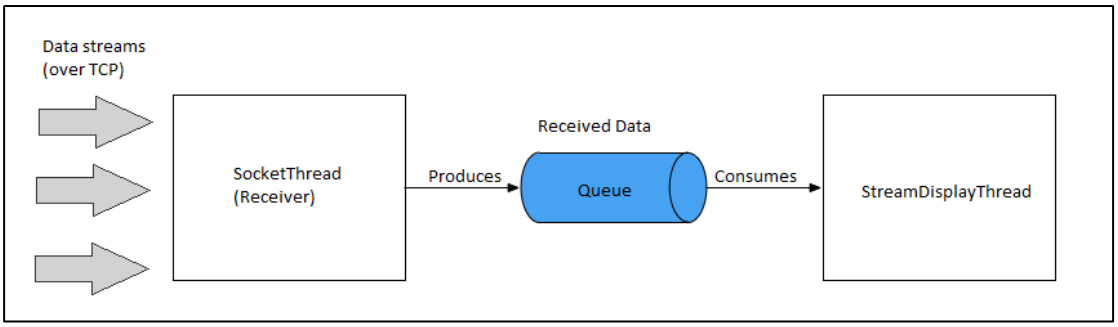
```
public static void send(String message) {
        printwriter.write(message + "\n");
        printwriter.flush();
}
```

As seen in the code above, the transmitter reads a complicated line, parses it and transmits a simple stream of <SampleId#SampleVal>.

This operation is done for every streamline in the streams file, and the transmitter assumes there's a receiver process listening to its streams.

6.2.2 Receiver

The receiver (as mentioned in chapter 5 – High Level Design) is composed of two main threads.



First, *SocketThread*, the thread responsible for the actual receiving of the data by setting up the other endpoint of the connection set by the transmitter. It listens on port 9191 to incoming data in the format <SensorId#Value>. On receiving data message, in case the logging option is set ON, it logs it into a log file (bmwLog) and then enqueues the data into the Producer-Consumer queue.

```
isr = new InputStreamReader(socket.getInputStream());
br = new BufferedReader(isr);
while((message = br.readLine()) != null){
	h.post(new Runnable() {
		@Override
		public void run() {
			Timestamp ts = new Timestamp(System.currentTimeMillis());
			Date d1 = new Date(ts.getTime());
			String logText = d1 + " " + message + "\n";
			if (AppState.isLogActive){
				FileOutputStream fileOutputStream =
						context.openFileOutput("bmwLog",
						Context.MODE_APPEND);
				fileOutputStream.write(logText.getBytes());
			}

		}
	});
	AppState.queue.add(message);
}
```

The Producer-Consumer queue is shared with the second thread, *StreamDisplayThread*, which is responsible for handling the data. It dequeues the data and handles the filtering and updating the graphs.

```
while (!AppState.queue.isEmpty() && running) {

    final String s = AppState.queue.poll();

    h.post(new Runnable() {
           @Override
           public void run() {
                  Timestamp ts = new Timestamp(System.currentTimeMillis());
                  Date d1 = new Date(ts.getTime());
                  final String id = s.split("#")[0];
                  final String val = s.split("#")[1];

                  if (!AppState.selectedIds.contains(id)){ //FILTERING
                         return;
                  } else {
                         final double double_val = Integer.valueOf(val, 16);
                         int graph_idx = AppState.selectedIds.indexOf(id);
                         int sid = AppState.selectedIds.get(graph_idx);
                         Sensor currSensor = AppState.getSensorFromId(sid);
                         double resolution = currSensor.getConfig().getResolution();
                         double offset = currSensor.getOffset();
                         double final_val = double_val * resolution + offset;

                         switch(graph_idx){
                           case 0:
                                series1.appendData(new DataPoint(d1,final_val),true,10,false);
                                break;
                           case 1:
                                series2.appendData(new DataPoint(d1,final_val),true,10,false);
                                break;
                           case 2:
                                series3.appendData(new DataPoint(d1,final_val),true,10,false);
                                break;
                         }
                  }
           }
    });
```

In the code above, while the queue is not empty and contains data to be parsed and viewed graphically, the thread enqueues the data, parses it into an id and a value, filters the data by the id (checking if the id was chosen earlier by the user). If the data is relevant, the corresponding graph shall be updated, otherwise ignore the data.

The ability to display the data graphically and in real-time graphs is obtained by the GraphView library.

See Appendix for GraphView library description and further information.

Also, see Appendix for sensors configurations file – the file that the application reads on startup to learn about the available sensors, and their default configurations.

7 Appendix

7.1 Stream File

The stream file as mentioned in section 6.2.1 - Transmitter, is a file produced by the vehicles system, and contains real-time information about the conditions and data collected by the sensors.

The stream file content is all Hexadecimal and is as follows:

File name: MW_samples_original_0001613_20181114_1754

```
Start DATA
MWSTRM 000000031001a10000SOH1002f0SOH100120387SOH
MWSTRM 0003ef021001100SOH1001202C4SOH
MWSTRM 0004a20210033011A39D30155236528CB1FFF0100SOH1001100SOH
MWSTRM 0005b4021001202D4SOH1001100SOH
MWSTRM 00075d021001202AESOH1001100SOH
MWSTRM 00088e0210033011A39D30155236528CC1FFF0100SOH10012029FSOH
MWSTRM 0009c8021001100SOH100120291SOH
MWSTRM 000b7a021001100SOH10012027CSOH
MWSTRM 000c750210033011A39D30155236528CC1FFF0100SOH1001100SOH
MWSTRM 000de702100120289SOH1001100SOH
MWSTRM 000f8802100120297SOH1001100SOH
MWSTRM 0010660210033011A39D20155236528CC1FFF0100SOH100120293SOH
MWSTRM 00120b021001100SOH100120297SOH
MWSTRM 00130802900118ec3SOH1004100017A7FSOH
MWSTRM 00131e041002a1027102710271027SOH1003d08CASOH100462710SOH1002a1027102710271027SOH
MWSTRM 00132b021003c7D000000SOH1004100017A7ESOH
MWSTRM 00133e021002a1027102710271027SOH100190100SOH
MWSTRM 00135a031002a1027102710271027SOH1003d08CASOH1004100027A7ESOH
MWSTRM 00136e021002a1027102710271027SOH1002700000000SOH
MWSTRM 001378031002a1027102710271027SOH100462710SOH1004100057A7FSOH
MWSTRM 001394041002a1027102710271027SOH1003d08CASOH1003c7D000000SOH1002a1027102710271027SOH
MWSTRM 0013aa02900118ec5SOH100320000548CSOH
MWSTRM 0013b40610014270FSOH100150000SOH1001100SOH1004100027A7FSOH1002a1027102710271027SOH
MWSTRM 0013d1021003d08CASOH1004100057A7FSOH
MWSTRM 0013dc031002a1027102710271027SOH100462710SOH1002a1027102710271027SOH
MWSTRM 001400021003c7D000000SOH100410001797FSOH
MWSTRM 00140b031002a1027102710271027SOH1003d08CASOH1002a1027102710271027SOH
MWSTRM 001430031004100047A7FSOH1002a1027102710271027SOH1002a1027102710271027SOH
MWSTRM 001449051003d08CASOH100462710SOH10033011A39D20155236528CD1FFF0100SOH900118ec5SOH10041000
MWSTRM 001460031002a1027102710271027SOH1003c7D000000SOH1002a1027102710271027SOH
MWSTRM 001483061004100027A7FSOH1002a1027102710271027SOH1003d08CASOH100120299SOH100320000548CSOH
MWSTRM 0014a6041004100017A7FSOH1002a1027102710271027SOH100462710SOH1002a1027102710271027SOH
MWSTRM 0014bb031003d08CASOH1003c7D000000SOH1004100027A7ESOH
MWSTRM 0014d3021002a1027102710271027SOH1002a1027102710271027SOH
MWSTRM 0014e902900118ec5SOH1004100017A7ESOH
MWSTRM 0014f4031002a1027102710271027SOH1003d08CASOH
MWSTRM 001511031002a1027102710271027SOH100462710SOH1004100047A7FSOH
```

Each streams file must contain a line indicating the start of the data samples ("`Start Data`"). In the example above, the first line indicates that the following lines are data samples.

Parsing of a stream file line is done according to the following structure:

Streamline:

MWSTRM AAAAAA BB C DDDD HHHH…HHH [SOH] DDDD HHHH…HHH [SOH]…

Parsing:

AAAAAA – 6 Hexadecimal digits representing the timestamp.

BB – 2 Hexadecimal digits representing the number of samples in the streamline.

C – 1 Hexadecimal digit representing sample type (if not equal to 1, then we ignore the line).

DDDD – 4 Hexadecimal digits representing the sensor ID.

H…HHH – Hexadecimal digits representing the sample's actual data.

SOH – Special character indicates an end of sample.

7.2 Sensors Configurations File

Sensors configurations file is a CSV file that the Android application (receiver) reads on startup to learn and get information about the available sensors, and their default configurations.

File name: sensors.csv

Content:

```
0001,Raw pressure(After LPF),Pa,0,1048575,0.01,0
0002,Slope,Deg,0,65535,0.1,500
0003,Height ASL,cm,0,1048575,1,0
0005,Real Odometer,m,0,4294967295,1,0
0007,Engine Hours,minute,0,4294967295,1,0
0008,Dashboard Odometer,m,0,4294967295,1,0
0011,Speed,km/h,0,255,1,0
0012,RPM,revolutions/minute,0,65535,1,0
0013,Pedal/Throttle,0~255,0,255,1,0
0014,Fuel Consumption Rate,L/1000km,0,65535,1,0
0015,Fuel Consumption Rate,L/1000hour,0,65535,1,0
0016,Engine Coolant Temperature,Deg,0,255,1,40
0017,Engine Running,boolean,0,1,1,0
0018,Engine Calculated Load,%,0,125,1,0
0019,Vehicle Weight(CAN),10kg,0,65535,1,0
001A,Vehicle Weight(MWProc),10kg,0,65535,1,0
001B,Vehicle Weight Percentage,%,0,100,1,0
001C,PTO State,boolean,0,1,1,0
001D,Cruise Control State,boolean,0,1,1,0
001E,Break Pedal State,boolean,0,1,1,0
0021,Gear state,0-neutral 251-parking,0,253,1,0
0022,Current retarder torque,No Unit,0,127,1,0
0028,Tank fuel level(CAN),%,0,255,0.4,0
0029,Raw Pressure(Before LPF),Pa,0,1048575,0.01,0
002F,Rough weight estimation,1-empty 3-full,0,3,1,0
0030,Low Resolution Fuelometer,mL,0,4294967295,0.5,0
0032,High Resolution Fuelometer,mL,0,4294967295,1,0
0038,Ambient temperature,0.01Deg °C +4000 offset,0,65535,0.01,4000
003A,Coarse weight estimation,No Unit,0,5,1,0
003D,Actual brake toque,Nm,0,32000,1,0
003F,Tire health,1-worst 10-best,0,10,1,0
0043,Supported OBD-II PIDs,No Unit,256,287,1,0
0044,Torque transfer percentage,%,0,100,0.4,0
0045,Streering position(CAN),Deg,-90,90.024645,0.002747,-90
0046,Vehicle CoG speed(CAN),Km/h,0,350,0.01,100
0047,Average rain intensity,%,0,100,1,0
0049,Supported OBD-II PIDs,No Unit,287,318,1,0
```

Each line is parsed as follows:

<SensorId>,<SensorName>,<Units>,<minVal>,<maxVal>,<Resolution>,<Offset>

7.3 GraphView Library

GraphView is a library for Android to programmatically create graphs and diagrams.

The GraphView library enables programmers to create Line Graphs, Bar Graphs, Point Graphs or create their custom graph.

Key Features

- Different plotting types: Line Chart, Bar Chart and Points Chart and they can be plotted together as a combination.
- Draw multiple series of data
- Let the diagram show more than one series in a graph.
- Realtime / Live Chart - Append new data live or reset the whole data.
- Secondary Scale.
- Tap Listener
- Handle tap events on specific data points.
- Show legend.
- Custom label formatter
- Handle incomplete data
- Viewport
- Scrolling and Scaling / Zooming
- XML Integration
- Optional Axis Titles
- Set vertical and horizontal axis titles.
- Customizable - color and thickness, label font size/color and more

XML Layout file:

```
<com.jjoe64.graphview.GraphView
        android:layout_width="match_parent"
        android:layout_height="200dip"
        android:id="@+id/graph" />
```

Java code:

```
GraphView graph = (GraphView) findViewById(R.id.graph);
LineGraphSeries<DataPoint> series = new LineGraphSeries<>(new
DataPoint[] {
          new DataPoint(0, 1),
          new DataPoint(1, 5),
          new DataPoint(2, 3)
});
graph.addSeries(series);
```

Reference: https://github.com/jjoe64/GraphView