Laboratory 5
Vehicle-Data

Agenda

Introduction

During this laboratory we will continue to extend our simulated vehicle functionalities. This time we will shift our focus from telematics to infotainment and instrument cluster.

Infotainment systems

Modern infotainment systems are capable to display relevant vehicle information and to provide the user with the possibility to access information and media using the car's head unit. With so many refinements in technology, infotainment systems in cars nowadays have become really advanced. With connected car techs like geo-fencing, smart navigation systems, voice-activated commands, Over-The-Air (OTA) updates and much more, infotainment systems are now more focused to make the drives secure as well as pleasurable. 7" to 12" touchscreens have become common in most of the cars nowadays. Android Auto and Apple CarPlay are also seen in most of the systems, which blends the smartphone into the vehicle ecosystem.

Instrument Clusters

Electronic instrument cluster is system designed to display a set of information to the driver. Instrument Clusters ensure that the driver is comprehensively and reliably informed at all times. They provide basic driving information like speedometer, tachometer, temperature, fuel, telltales, and warnings. What was once a piece of plastic with drilled holes into which gauges and instruments were screwed in, has now become a real high-tech object. Instead of analog dial-type gauges the modern instrument clusters are powerful and intuitive and they use digital displays.

Basic Features

As you may already know from course 3, infotainment and instrument cluster features comprise many practical applications, such as:

- Human Machine Interface
  - GUI
  - Speech Recognition

    - Text to speech – eg: reading SMS, social media posts etc.
- Radio (AM, FM / DAB / SDARS)
- Media Player (CD / DVD / etc.)
- Weather
- Navigation
- Phone
    - Apple CarPlay / Android Auto -> Replicate phone display on dashboard
    - Hands-free phone connection – using Bluetooth
- Information about your car
- Cloud Terminal
- Actuated Sound
- Entertainment – movies, games, social entertainment etc.
- Rear View Camera

Practical application overview

For the first part of our practical application, we are going to focus on the feature “Information about your car”. All cars are equipped with sensors that are able to collect live data about operational parameters. We are going to implement an application that is going to simulate this behaviour of the instrument cluster. We will gather data and display it to the driver using an user interface, for example :battery level, coolant temperature, transmission fluid level, tire pressure, engine oil pressure.

For the second part, we will try to simulate the remote diagnosis behaviour of modern vehicles. Cars emit numerous trouble codes that can be used to check vehicle health, accelerate repairs, and predict when a vehicle needs to be serviced. Usually the car is capable of warning the user about certain problems. For instance if the oil level is low, a “low oil pressure” warning light will appear to notify the user. If there is an issue with the oxygen sensor a yellow “check engine” warning will appear that tells the driver he needs to check the engine as soon as possible. However, with each make, model, and onboard device generating a different set of codes in different formats, determining the exact issue is not so easy. The driver can’t determine the exact issue or take action until the vehicle arrives at a servicing facility. There, the authorized service personnel will connect to the OBD-2 interface of the car and will be able to read the DTC codes that were stored by the car. A DTC, short for Diagnostic Trouble Code, is a code used to diagnose malfunctions in a vehicle. The vehicle generates it when detecting a problem. Below you can find a short description of how DTC codes are formed.

The first character (number)

DTC codes start with a letter that denotes the part of the vehicle that has a fault.

P – Powertrain. Includes engine, transmission, and associated accessories.

C – Chassis. Covers mechanical systems and functions: steering, suspension, and braking.

B – Body. Parts mainly found in the passenger compartment area.

U – Network & vehicle integration. Functions managed by the onboard computer system.

The second character (number)

The first letter is followed by a number, usually 0 or 1.

0 – Standardized (SAE) code, also known as generic code (sometimes called global)

1 – Manufacturer-specific code (sometimes called enhanced)

The third character (number)

For powertrain codes, this number tells you which vehicle subsystem has a fault. There are eight:

0 – Fuel and air metering and auxiliary emission controls

1 – Fuel and air metering

2 – Fuel and air metering – injector circuit

3 – Ignition systems or misfires

4 – Auxiliary emission controls

5 – Vehicle speed control, idle control systems, and auxiliary inputs

6 – Computer and output circuit

7 – Transmission

The fourth and fifth characters (number)

The final piece of a DTC is a number that defines the exact problem that you're experiencing. It can be a number between zero and 99.

Here's an example of a complete code:

P0782 means powertrain, generic, transmission, 2-3 shift malfunction.

As we describer earlier, most of the vehicles require a connection to the OBD-2 interface of the car to be able to read these error codes. Also, intensive knowledge/experience about that specific model of vehicle is required in order to be able to interpret the DTC correctly and be able to solve the issue. This is where the remote diagnostic part comes in handy. Through remote diagnostics service, the driver is able to monitor the health of the vehicle in near real time. He can easily monitor a range of conditions that could take a vehicle out of service or increase the operation cost.

For remote-diagnostics capable cars, the car is continuously sending data to the backend server of the car manufacturer. The data is comprised of various operational parameters of the car, such as sensor reads, but also stored DTC codes. The backend is able to quickly process the data and determine if immediate action is needed. Usually the user can be informed of certain issues by notifications arriving on the infotainment unit.

Preconditions

Software to install:

Python : https://www.python.org/downloads/windows/ (please select Latest Python 3 Release)

Pycharm: https://www.jetbrains.com/pycharm/download/#section=windows

Already available items:

a) a script called **diagnosisClient.py** which implements a TCP client

b) a script called **diagnosisServer.py** which implements a TCP server

c) a json file **carInformation.json** containing a set of car information

Hints

In order to run client/server, you can run the following command.

py diagnosisClient.py 127.0.0.1 -p 65432

py diagnosisServer.py 127.0.0.1 -p 65432

Address and port are specified as command line parameters. If both scrips are run successfully you should be able to see two windows as shown in the image below.

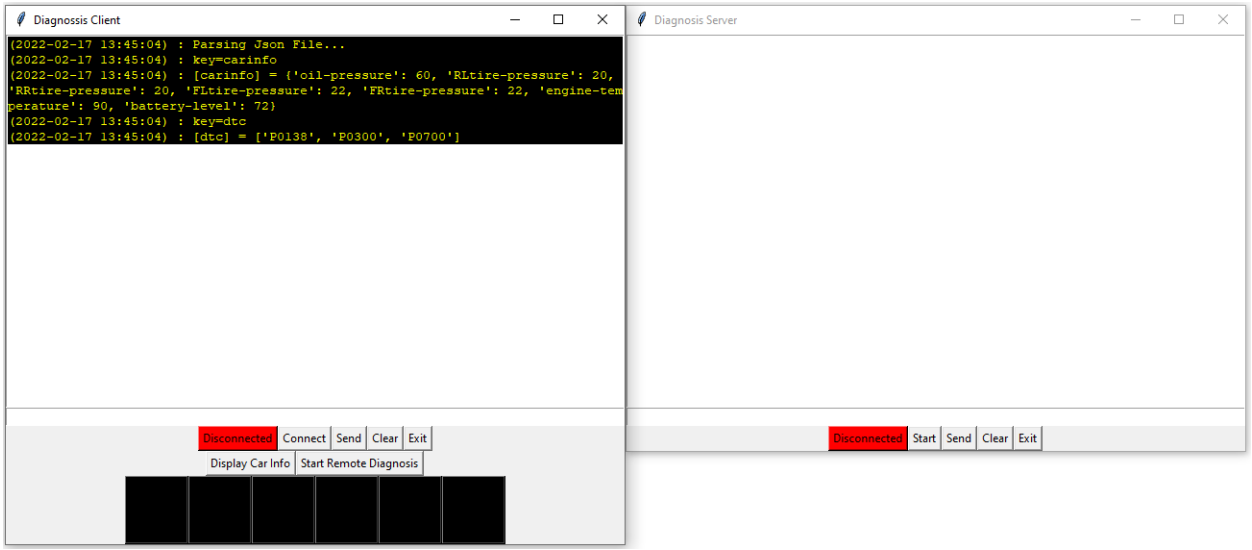

Figure 1: Client & Server – graphical view

Both the client and the server have the following elements:

- text box: all messages server/client are printed
- input box: collects user input

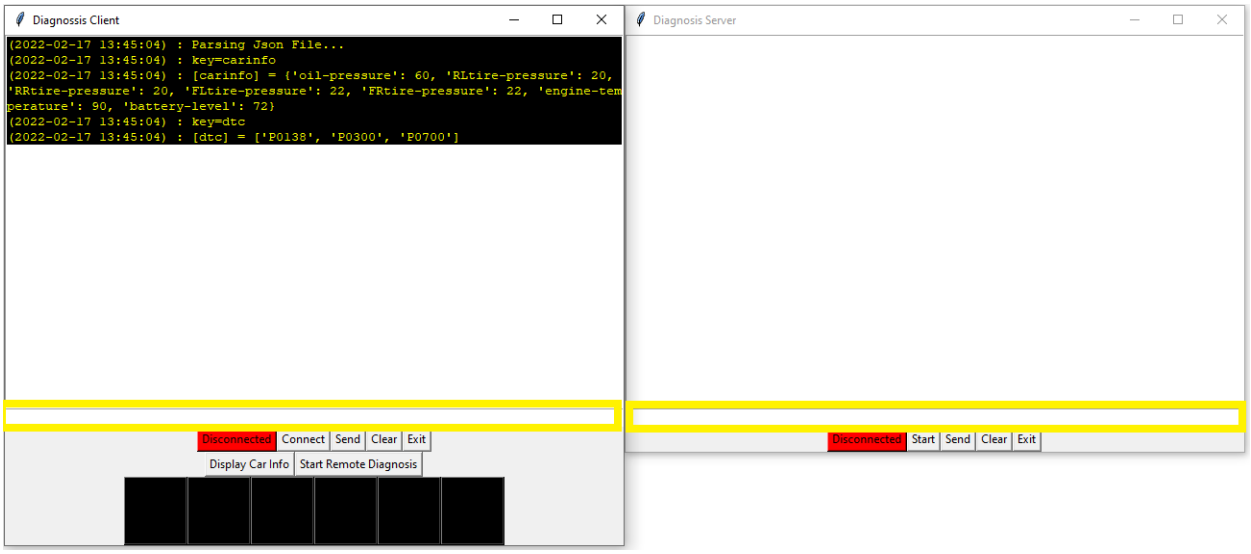

Figure 2: Input box (highlighted with yellow) → For commands entered by user

- buttons:

  connect (client only): client connects to the server

  start (server only): server starts listening for tcp connection

  send: attempts sending the data from input box to server/client

  clear: clears text box (messages shown)

  Display Car Info: should print a popup that displays all the information about the vehicle

  Start Remote Diagnosis: should send all the stored dtc codes to the server

  exit: exit application

Steps to connect client to server:

1. Server -> Start
2. Client -> Connect

You should see the red button turning green.

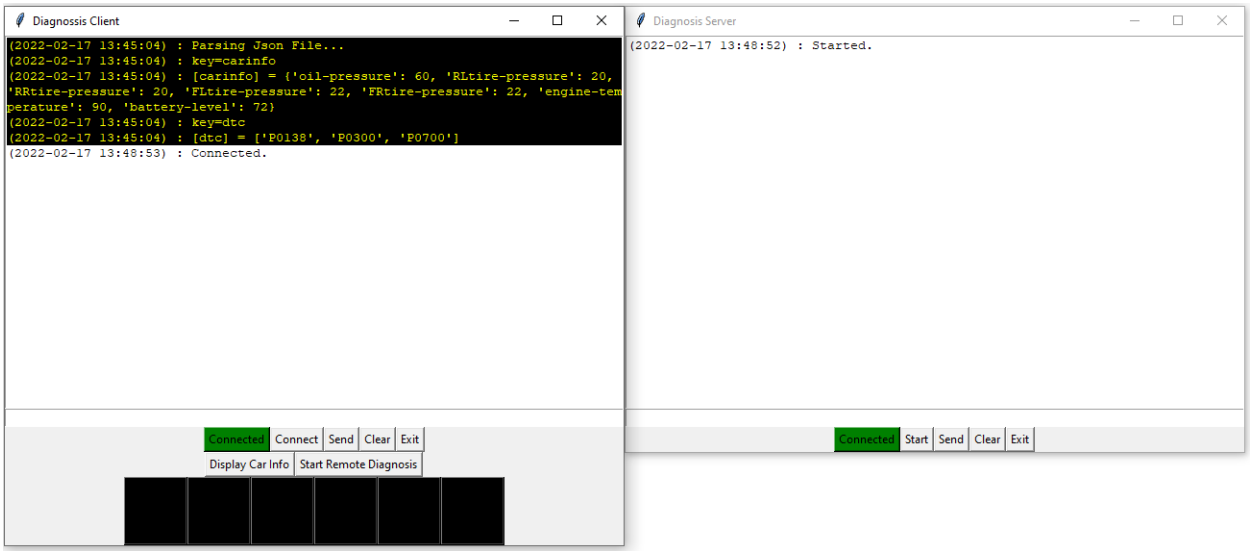

Figure 3: Client & Server – successful connection

At startup, the server is parsing the file carData.json and saves data into two separate class members named car_data and dtc_list.

car_data is a list that contains information about the vehicle.

```
(2022-02-14 20:35:23) : [carinfo] = {'oil-pressure': 60, 'RLtire-pressure': 20,
'RRtire-pressure': 20, 'FLtire-pressure': 22, 'FRtire-pressure': 22, 'engine-tem
perature': 90, 'battery-level': 72}
```

dtc_list is a list that contains all the dtcs stored by the vehicle.

```
(2022-02-14 20:35:23) : key=dtc
(2022-02-14 20:35:23) : [dtc] = ['P0138', 'P0300', 'P0700']
```

After the json file is parsed, the diagnosis client, will automatically send the list of dtcs to the diagnosis server. Based on the received dtc list, the server will send a set of actions to the client. The possible actions which client can receive from server are:

display-popup – server sends this command in order for client to show a notification to the driver

service-soon – server sends this command in order for the client to turn on the service-soon warning light.

Tasks

a) In order to complete this you will need to modify method __init__ from class client. As mentioned above, car_data is a list that contains all the information that is read from carData.json file. The data can be displayed in user interface by default by pressing the "Display Car Info" button. What you need to do is to modify __init__ method to create this behaviour:

If oil pressure < 30 or oil pressure > 60, then turn on the oil pressure light

If pressure of front tires < 20 or pressure of front tires > 24, then turn on the tire pressure light.

If pressure of rear tires < 18 or pressure of front tires > 22, then turn on the tire pressure light.

If engine temperature > 90, turn on the engine temperature light.

If battery-level < 15, turn on the battery level light.

You can use the following methods that are already implemented in order to switch the lights.

```
def turn_check_engine_light(state):
def turn_oil_pressure_light(state):
def turn_tire_pressure_light(state):
def turn_engine_temperature_light(state):
def turn_battery_light(state):
def turn_service_soon_light(state):
```

b) In order to complete this, you need to modify the handle_message method of the diagnosis client. You need to change the behaviour under the following condition.

```
if command.startswith('display-popup'):
```

When display-popup notification is received from server, create a popup notification for the user that display the entire message sent by the server. You can use auxiliary method print_popup in order to display the message to the driver.

```
def print_popup(self, message):
```

c) In order to complete this, you need to modify the handle_message method of the diagnosis client. You need to change the behaviour under the following condition.

```
if command.startswith('service-soon'):
```

When service-soon notification is received from server, turn on the service soon light on the virtual dashboard.

Evaluation

Score:

a) 1.5 point(s)

b) 2 point(s)

c) 1.5 point(s)