Architecture of on-line data acquisition system for car on-board diagnostics

Bartosz Kowalik[1,*] and *Marcin Szpyrka*[1]

[1]AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering, Department of Applied Computer Science, al. A. Mickiewicza 30, Poland

Abstract. Modern cars produced for the last two decades are full of electronic devices called Electronic Control Units (ECU). They are responsible for collecting diagnostic data from different components such as the engine, breaks *etc.* using probes and sensors. The collected data are validated against built-in heuristic and abnormal behaviour is reported to a driver by a gauge on an instrument cluster. ECUs use data provided by other ECUs. Information is transmitted over the dedicated network called Controlled Area Network (CAN). Every car equipped with ECUs and CAN exposes information over universal diagnostic interface called On-Board Diagnostic. Using the interface, it is possible to gather car's live data. With the data mining approach, it is possible to exploit the collected more effectively to obtain much more information about the functioning of car components than it is provided by standard vehicle equipment. The paper describes how to build a laboratory set to facilitate automated data collection. It consists of three major components: data acquisition, automated logs collection and persistent storage with presentation tools. The first component is based on Torque application for which reverse engineering was performed.

1 Introduction

The paper deals with the problem of data acquisition from the car's ECUs and preparing them for further exploration with data mining techniques. The conclusion from the analysis of available literature in the field was that there was no automated solution to capture the live stream of data from a car and store it. The key element of further research was to develop a tool that would collect data while the car is working and in a way that would not engage the driver. A car can produce tons of information every second. Its main capabilities are:

- high throughput,
- resilience.

The tool is expected to be ready for live data analysis. It should allow detecting faulty components and potentially preventing unexpected car breakdown. Some components do not show signs of excessive wear prior to damage, therefore, it is hardly possible to prevent a car going out of service. Nevertheless, historical data analysis in such a case can explain what had happened.

2 Architecture

The key part of our solution was the designing phase. As it was already said, the system must work on-line while a car is working. Moreover, the system cannot require interaction with the driver. One of the most important requirements was the hardware used so far [8]:

- Xiaomi Redmi 4X – an Android-based smartphone,
- ELM372 OBD2 - Bluetooth OBD2 interface.

The set has been extended with an HTTP server that is crucial for data processing. Main architecture parts of the considered system are presented in the following diagram.

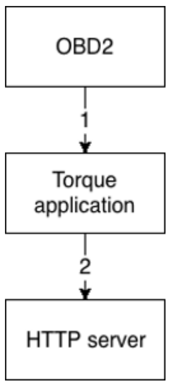

Fig. 1. Architecture overview.

2.1 Torque

Torque is an Android application which collects live data from a car. It uses the OBD2 Bluetooth device to which the application connects. Torque sends data over the Internet to an external server. By default, it sends data to a Torques' developers web server, which has an online lookup panel. In the application options, it is possible to override target URL with a custom one.

Developers have not provided any documentation for the communication protocol. In order to implement a custom server, the protocol was reverse engineered. This

[*] Corresponding author: bkowalik@agh.edu.pl

process required multiple steps, which are described in this section.

In the first step, there was an instance of the server that would log every incoming request to a file. After that, multiple test drives were performed to collect sample logs.

In the next step, every single URI parameter had been analysed to decode its meaning. Every parameter encodes the key-value pair. Torque developers provide basic documentation for Torque plug-in developers [9], and that the documentation was used to guess the meaning of some parameters. It contains identifiers (keys) with description organised in a table. OBD2 PIDs are usually presented in the hexadecimal format and Torque developers followed the same convention. Some parameters mentioned on the website are similar to the ones, which are sent to the server. That lead to the conclusion, letter *k* means *0x* in hexadecimal encoding. For instance, *kc* actually is *0x0c* and based on OBD2 PIDs list [10] means engine revolutions per minute descriptor. For several of them, it was possible to decode their meaning-based names and values:

- eml - email address,
- v - version,
- session - milliseconds since January 1, 1970 UTC, identifies when measurements started,
- id - session identifier, after broad analysis it turned to be a string representation of a UUID [11] with stripped "-",
- time - milliseconds since January 1, 1970 UTC, identifies single measurement precise measure time

For testing purposes, there was a special script developed which parses log files and makes fake server calls. After testing it turned out that there were still certain unknown parameters. There was no information about them either in Torque documentation or OBD2 PIDs list.

Inspecting the application directory on Android smartphone revealed a file *torqueConf.dat*. Its extension indicated that information stored there could be binary encoded but it was not. is the file can be opened in any text editor and content of a file is editable. A part of the file's content is presented in the listing [1].

Listing 1. torqueConf.dat

```
log_pid19=16732675
log_fullName9=Acceleration Sensor(X axis)
log_pid18=16732674
log_fullName8=Acceleration Sensor(Total)
```

Having analysed *torqueConf.dat* file's content it was clear that very high integer numbers, such as *16,732,675* or *16,732,674* are in fact PIDs. That is because Torque stores custom PIDs in decimal form, which is different from what can be found in official documentation. The next step was to convert all missing PIDs from hexadecimal numbers to decimal ones. Having done that, *torqueConf.dat* file was used as a lookup. This configuration file has the key value structure. Decimal PID number is a value, which means that part of a key has an internal ID of a measured value. For example, PID *kff5203* is *0xff5203* hexadecimal and *16,732,675* decimal. The value 16,732,675 is pointed out by the key *log_pid19*. That means, it indicates internal ID, which is 19. Following the pattern, *log_fullName19* indicates a description of a measurement. In this example, it should be read as *Litres Per 100 Kilometer(Long Term Average)*. Using that technique, the following PIDs were decoded:

- kff126b- remaining fuel calculated,
- kff126a - distance to empty,
- kff1273 - engine kW at wheels,
- kff1272 - average trip speed,
- kff1271 - fuel used trip,
- kff125d - fuel flow rate hour,
- kff129a - android device battery level,
- kff1204 - trip distance,
- kff1267 - GPS bearing,
- kff1266 - trip time since journey start,
- kff1269 - volumetric efficiency calculated,
- kff1268 - trip time whilst moving,
- kff5203 - litres per 100 kilometres,
- kff5202 - kilometres per litre,
- kff5201 - miles per gallon.

2.2 HTTP server

The HTTP server plays a key role: it has to keep up with the high load generated by live data uploaded from Torque. For that reason Akka HTTP framework [12] was used. According to the documentation it is advised as high-throughput middleware.

Torque uses URI (Uniform Resource Identifier) to send recorded parameters like the one presented in listing [2].

The first implementation was not capable of handling long URIs [13] and was returning 414 HTTP status code [14]. After maximum URI length had been increased, the server was capable of handling long URIs.

Subsequently, the proper application programming interface was defined. The server accepts only GET requests. This process is called dispatching. Akka HTTP uses the dispatchers thread pool. This does not mean that there is a thread per each incoming connection – these are only processed requests that require threads to be processed.

Processing a request means that it is first processed and then both published on the message queue and saved to a database. Input and output operations are in general much slower compared to application's code execution. The response is sent back to a client only if a database call finishes, but the message publishing process is asynchronous.

3 Integration points

This section describes external services used in order to either store or move live data further.

3.1 Message broker

The described HTTP server, apart from storing new data in the database, also produces new records on the message queue.

Listing 2. Torque URI sample

```
?eml=bartekviper@gmail.com&
v=8&session=1521995741941&
id=80c9065bff55006e9f5f3d4f8d9456ae&
time=1521996644906&kff1005=19.96196812&
kff1006=50.08527848&
kff1001=27.504&
kff1007=275.4&
kff129a=41.0&
k2d=99.21875&
k33=98.0&
kb=128.0&
k23=36700.0&
kff1267=232.387&
kff1268=577.541&
kff1266=809.0&
kff1239=6.068&
kff1269=92.0&
kff1005=19.96196812&
kff1006=50.08527848&
kff123b=275.4&
kff1203=2.646776&
kff5202=5.7647285&
kff5203=17.34651&
kff1201=7.4767685&
kff5201=16.284544&
kff1226=8.839944&
kff1273=6.591945&
kff1225=17.75426&
kff1238=14.1&
k42=14.349&
k4=0.0&
kc=2257.0&
k21=0.0&
k31=65535.0&
kff126b=40.63311&
kff126a=692.2412&
kff1204=4.1081877&
k10=38.25&
kff1001=27.504&
kd=32.0&
kff1237=1.4079971&
kff123a=14.0&
k2c=4.7058825&
k46=12.0&
kf=26.0&
k5=84.0&
kff1202=4.3511324&
kff1206=3.6139567&
kff1010=250.0&
kff1271=1.5658529&
kff125d=13.979028&
kff125a=232.9838&
kff1272=21.435392&
kff1208=27.669928
```

In order to achieve high throughput, Apache Kafka is used [15]. There are three main parts:

- producer,
- consumer,
- topic.

Topic is essentially a message queue. A new message is stored with an associated key. The key can be *null,* which means that there is none. The messages can be arranged logically and determine to which partition messages belong to if a distributed mode is enabled. Each partition has its own message ordering based on a key. They are responsible for physical message arrangement.

Producer is an element which publishes new messages to a topic. It sets message content and associates key with it.

The consumer reads messages from the topic. Everyone is a member of the consumer group. The message is consumed once read in a single consumer group. Each consumer has an offset which determines what messages were read and are about to be read.

The HTTP server produces messages to a single topic called *raw_car_data*. Every measurement type such as for example engine coolant temperature is published with an individual key. Each published message is stored using JavaScript Object Notation (JSON) [16] format. This format does not impose a strict schema. A message example is presented in listing [3].

Listing 3. Torque URI sample

```
{
    "timestamp": 1531427308000,
    "session": "2018-07-12T22:28:28+02:00",
    "id": "05d90cdf-b2f6-45d9-aafe-ffb96da96562",
    "name": "gps-bearing",
    "value": 51.0
}
```

Fields are:

- timestamp - milliseconds since January 1st 1970,
- session - ISO 8601 format date and time [17],
- id - unique session identifier,
- name - parameter name,
- value - parameter value.

3.2 Persistent storage

Persisting data is key for historical data analysis. Stored information does not have a relational structure and therefore NoSQL database was selected. There is a wide selection of different NoSQL databases depending on a purpose, such as for example:

- graph,
- big table,
- time series.

InfluxDB time series database was selected [18] for the considered system. Data are stored in a structure called *measurement*. Each record has a key, which is a timestamp of a measurement, fields that contain values and tags. The application stores every measured parameter obtained from different measurements. A session identifier is associated to every record as a tag, which allows such aggregations as average.

4 Data analysis

This section describes the initial analytic method employed in the study as well as, how the application is designed for the future live operation.

4.1 Parameterised entropy

Entropy is a measure of uncertainty. It can be used to summarize feature distributions in a compact form, for example, a single number. Many forms of entropy exist, including Shannon entropy [1], T-entropy [2], parameterized Tsallis entropy [3], [4], parameterized Renyi entropy [5], [6] *etc*. We focus on the last two entropies due to their highly interesting features. The Shannon entropy assumes a tradeoff between contributions from the main mass of the distribution and the tail. If the parameterised Tsallis or Renyi entropy are used, we can control this tradeoff [7].

An assumed discrete random variable *X* is given. In information theory, entropy is a measure of the uncertainty associated with the random variable *X*. The more random the variable *X*, the bigger the entropy of *X*.

The Tsallis entropy is defined by:

$$H_{T\alpha}(X) = \frac{1}{1-\alpha}\left(\sum_{i=1}^{n} p(x_i)^{\alpha} - 1\right) \qquad (1)$$

where $p(X = x_i)$ is the probability distribution of *X*.

The Renyi entropy is defined by:

$$H_{R\alpha}(X) = \frac{1}{1-\alpha}\log_{\alpha}\left(\sum_{i=1}^{n} p(x_i)^{\alpha}\right) \qquad (2)$$

Both entropies use the α parameter to control the trade-off. If the parameter has a positive value, it exposes the main mass, if the value is negative – it exposes the tail. The best value of the α parameter is chosen from data experiments.

4.2 Live entropy computation

Live data analysis is the key element of this application. It has a built-in support for publishing information on Kafka topic, which makes it extendable by writing in any programming language Kafka messages consumer.

Using this mechanism, parameterised entropy computation is implemented. For a start only for engine coolant temperature. In order to implement that measurement on live stream data windowing technique will be used. Entropy is calculated against the circular buffer. The time window is moved by certain delta and the procedure is repeated. The final result is stored in the database and can be published on another topic for further use.

A part of the collected samples are both working and damaged thermostat data.

Figures 2 and 3 show the thermostat behaviour over time. The characteristics of thermostat which works are presented in Figure 2. It maintains in this case the engine temperature within a certain range. On the contrary, Figure 3 presents a damaged thermostat which exhibits unstable characteristics: it is not capable of maintaining the engine in its operating temperature range.

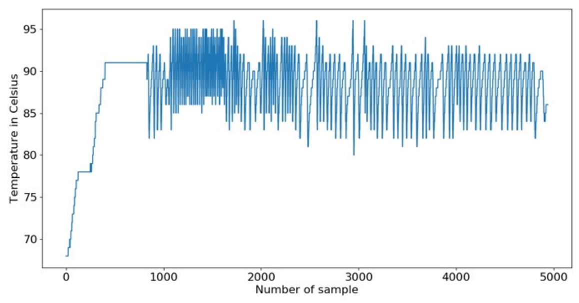

Fig. 2. Normal working thermostat.

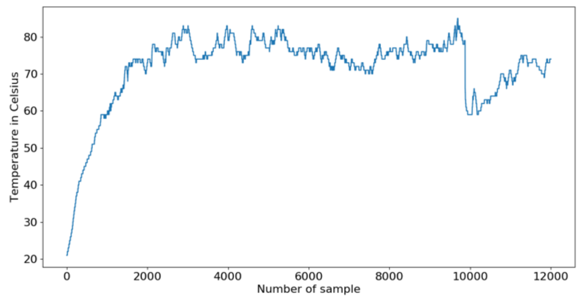

Fig. 3. Damaged thermostat.

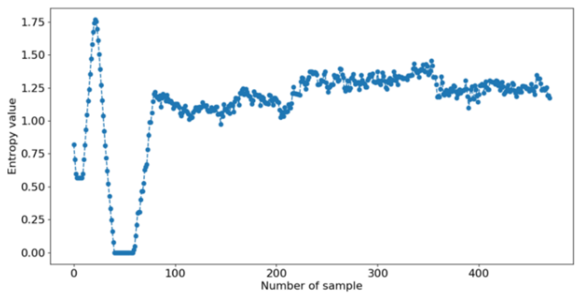

Fig. 4. Renyi entropy for the working thermostat.

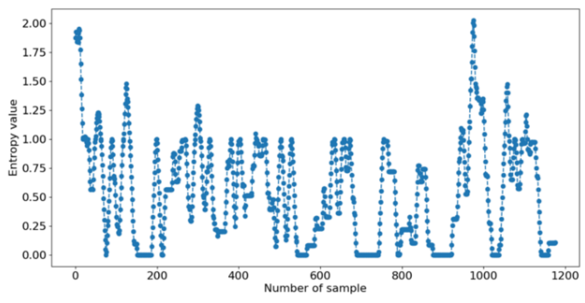

Fig. 5. Renyi entropy for the damaged thermostat.

For those two examples, Renyi entropy was calculated and presented in Figures 4 and 5. Figure 4 shows Renyi entropy for a working thermostat, Figure 5 for the damaged one. It was calculated for parameter α=4.

5 Conclusion

Based on the research, it was settled that thus far there has been no existing platform which would enable the car's live data collection and analysis. Our solution is capable of storing and further publishing measurements in both offline batch analysis and online processing. Through open source technologies and standards, it is possible to extend the scope of data handled to incorporate other tools, written in any programming languages.

Parameterised entropy analysis provided more information about detecting an upcoming failure. Although in the research data from one car had been used, there is a high possibility that it would work for any other as well, and it would furthermore require adjusting the original parameters.

Future works are planned to increase the efforts to improve the platform, which will include, *e.g.* the implementation of online entropy computing. Also, there is a great need for developing the platform documentation.

References

1. Shannon, C.: A Mathematical Theory of Communication. Bell System Technical Journal, 1948, Vol. 27, pp. 379-423.
2. Titchener, M.R.; Nicolescu, R.; Staiger, L.; Gulliver, T.A.; Speidel, U.: Deterministic Complexity and Entropy. Fundamenta Informatica, 2005, vol. 64, pp. 443-461.
3. Tsallis, C.: Possible generalization of Boltzmann-Gibbs statistics. Journal of Statistical Physics, 1988, vol. 52, pp. 479-487.
4. Prehl, J.; Essex, C.; Hoffmann, K.H.: Tsallis Relative Entropy and Anomalous Diffusion. Entropy, 2012, vol. 14, pp. 701-716.
5. Renyi, A.: Probability theory. By A. Renyi. [Enlarged version of Wahrscheinlichkeitsrechnung, Valoszinusegszamitas and Calcul des probabilites. English translation by Laszlo Vekerdi]; North-Holland Pub. Co Amsterdam, 1970.
6. Csiszár, I. Axiomatic Characterizations of Information Measures. Entropy, 2008, vol. 10, pp. 261-273.
7. Bereziński, P.; Jasiul, B.; Szpyrka, M.: An Entropy-Based Network Anomaly Detection Method. Entropy, 2015, vol. 17, pp. 2367-2408
8. Kowalik B.: Introduction to car failure detection system based on diagnostic interface. 2018 International Interdisciplinary PhD Workshop (IIPhDW), 2018, pp 4-7
9. PluginDocumentation. https://torque-bhp.com/wiki/PluginDocumentation Accessed: 2018-06-20
10. OBD-II PIDs. https://en.wikipedia.org/wiki/OBD-II_PIDs Accessed: 2018-06-20
11. A Universally Unique IDentifier (UUID) URN Namespace. https://tools.ietf.org/html/rfc4122 Accessed: 2018-06-20
12. Akka HTTP documentation. https://doc.akka.io/docs/akka-http/current/index.html?language=scala Accessed: 2018-06-20
13. Fielding R.; Reschke J.: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. http://www.rfc-editor.org/rfc/rfc7231.txt Accessed: 2018-06-20
14. Fielding R.; Reschke J.: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. http://www.rfc-editor.org/rfc/rfc7230.txt Accessed: 2018-06-20
15. Apache Kafka https://kafka.apache.org/documentation Accessed: 2018-06-20
16. The JavaScript Object Notation (JSON) Data Interchange Formats https://tools.ietf.org/html/rfc8259 Accessed: 2018-06-20
17. Date and Time on the Internet: Timestamps https://tools.ietf.org/html/rfc3339 Accessed: 2018-06-20
18. InfluxDB http://docs.influxdata.com/influxdb/v1.5/ Accessed: 2018-06-20