FACULTY
OF ELECTRICAL
ENGINEERING
CTU IN PRAGUE

Master's Thesis

# Data Acquisition and Control Framework for an Intelligent Vehicle

***Bc. Jan Nejtek***

Department of Measurement
Supervisor: doc. Ing. Jiří Novák, PhD.

4 January 2022

# MASTER'S THESIS ASSIGNMENT

## I. Personal and study details

Student's name: **Nejtek Jan** Personal ID number: **466128**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Measurement**

Study program: **Open Informatics**

Specialisation: **Computer Engineering**

## II. Master's thesis details

Master's thesis title in English:

**Data acquisition and control framework for intelligent vehicle**

Master's thesis title in Czech:

**Framework pro sběr dat a řízení inteligentního vozu**

Guidelines:

Design and implement a data broker middleware framework for communication between the hardware (vehicle) dependent layer providing vehicle status information and independent vehicle control, and model layer providing prediction of vehicle users behavior.
Middleware should be able to communicate with hardware dependent layer using a dedicated REST API. With it own REST API it will provide an interface for predictive models.
Analyze the requirements, choose suitable software platform, design and justify the structure of your implementation.
Design a sutable form of communication between middleware and predictive models.
Implement complete system and verify its functionality using a specific use case and respective predictive model (e.g. for adaptive cruise control/speed limiter functionality).

Bibliography / sources:

Beran, J.: Firmware for Control Module of an Intelligent Vehicle, Diplomová práce ČVUT FEL, 2020

Name and workplace of master's thesis supervisor:

**doc. Ing. Jiří Novák, Ph.D., Department of Measurement, FEE**

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **13.09.2021** Deadline for master's thesis submission: **04.01.2022**

Assignment valid until:
**by the end of winter semester 2022/2023**

| doc. Ing. Jiří Novák, Ph.D.<br>Supervisor's signature | Head of department's signature | prof. Mgr. Petr Páta, Ph.D.<br>Dean's signature |
|---|---|---|

## III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

| . | |
|---|---|
| Date of assignment receipt | Student's signature |

# Acknowledgements

I would like to express my sincere gratitude to my supervisor doc. Ing. Jiří Novák, PhD. for his continued support, guidance and mentorship. I would also like to thank Ing. Jan Sobotka, PhD. for his aid in establishing the functionality of the initial prototypes of the devices used in my thesis.
Furthermore, I would like to thank Ing. Jaroslav Beran, whose diploma thesis called *Firmware for Control Module of an Intelligent Vehicle* provided an important component for my framework – the Controller Area Network (CAN) Activator.
I also want to thank my dear friend Anne Vedel Hansen for her help with proofreading and correcting the language used in this thesis. Last but not least, I have to thank my dog Benjamin for providing unconditional support during my studies and work on this thesis.

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

Prague, 4 January 2022 .....

Czech Technical University in Prague
Faculty of Electrical Engineering


### Citation of this thesis

Nejtek, Jan. *Data Acquisition and Control Framework for an Intelligent Vehicle*. Master's Thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, 2022.

# Abstrakt

Cílem této práce je navrhnout systém pro sběr dat z komunikačních sběrnic vozidla a pro následné ovládání funkcí tohoto vozidla. Tyto funkce jsou zpřístupněny okolnímu světu pomocí HTTP(S) REST API za účelem propojení s externími modely strojového učení / umělé inteligence. Celkový účel této soustavy je pro zajištění zvýšení pohodlí a bezpečnosti řidiče a pasažérů vozidla pomocí předvídání různých akcí, jako je nastavování klimatizace, vyhřívání sedadel nebo adaptivního tempomatu, a jejich provádění bez nutnosti ztráty pozornosti řidiče na dobu delší než nezbytně nutnou.

Práce je rozdělena do čtyřech hlavních částí. První část popisuje architekturu systému a diskutuje jeho rozdělení do jednotlivých komponent. Druhá část zkoumá požadavky na zmíněné komponenty a popisuje jejich návrh. Následně třetí část zkoumá spolehlivost a analyzuje možná selhání těchto komponent. Čtvrtá část popisuje reálnou vestavbu systému do vozidla a k tomu potřebnou konfiguraci hardware a software.

**Klíčová slova** sběr dat; REST API; vestavný systém; ovládání vozidla; CAN; ASP.NET Core; Flask; Qt

**Překlad názvu** Framework pro sběr dat a řízení inteligentního vozu

# Abstract

The goal of this thesis is to design a framework for the acquisition of data from a vehicle's systems and for the control of the same vehicle's functionality. This functionality is then exposed to the outside world via a HTTP(S) REST API in order to interface with external machine learning / artificial intelligence models. Together, the purpose of this system is to further the comfort and safety of the vehicle's occupants by predicting certain actions, such as adjusting the air conditioning, seat heating, or adaptive cruise control, and carrying them out without the need for the driver to lose focus for extended amounts of time.

This thesis is split into four main parts. In the first part, the architecture of the whole framework is discussed, with the reasoning behind its separation into individual components. The second part explains the requirements and the design of individual components in the framework. Subsequently, the third part investigates reliability and failure analysis of said components and the mitigation of errors. The fourth part explains the physical implementation of the framework in the vehicle, and the necessary hardware and software configuration.

**Keywords** data acquisition; REST API; embedded system; vehicle control; CAN; ASP.NET Core; Flask; Qt

# Contents

# List of Figures

# List of Tables

Chapter 1

# Introduction

Recent years have seen an interesting phenomenon. As more and more comfort features have been added to passenger cars, the amount of buttons for the occupants to push has regressed. That is, of course, easily explained by the use of modern day touch screen interfaces and similar technology, such as touch pads or scroll wheels. This, however, raises the matter of safety concerns as the attention of the driver can be needlessly broken by trivial actions regarding the control of the car's comfort functions that used to rely on muscle memory and touch feedback, which now require one or more interactions with a flat touch screen.

Even when the driver fully focuses on the road and the interactions with the car's comfort system are handled by the front **passenger**, a potentially unsafe situation can occur in case of misunderstandings and errors when adjusting the controls, such as turning the sound system volume way up instead of turning it down, or the passenger getting angry at (subjectively) nonsensical controls or suboptimal touch screen sensitivity.[1]

A solution has been proposed which eliminates this element of failure by shifting the difficult[2] parts of control (which element to control and what value to set) to an artificial intelligence model and leaving the user with a simple question (ex. "Would you like to switch the radio sound source to Bluetooth?") and a selection between Yes and No.

This setup also has the possibility to make the vehicle more comfortable and approachable for first-time, or otherwise inexperienced drivers and passengers (e.g. rental or company-owned cars) and thus increase the overall satisfaction with it. Even compared to the vehicles of yesteryear which relied on traditional muscle memory and tactile feedback, there is no learning curve with the use of a machine learning-aided decision mechanism, and there is further potential for voice control and text-to-speech integration.

[1] Author's personal experience with family members.
[2] Not meant as "difficult to decide", but more as "difficult to correctly input".

Such a concept has already been popularized as far back as 1982 by action crime drama television series Knight Rider, whose protagonist, crime fighter Michael Knight, is assisted in his adventures by a car named KITT[3]. This car shows features discussed in above paragraphs, such as artificial intelligence, voice control, and an artificial voice of its own. Furthermore it has the ability to self drive and a high tolerance to damage, features that are not a part of the topic of this thesis.

Figure 1.1: Pontiac Firebird (third generation) stylized as a replica of the KITT vehicle. Image used with permission. Photo 176286875 © Dawid Kalisinski | Dreamstime.com

The entertainment franchise started by the original television series has grown considerably in the following years, adding two more television series released as recently as 2009, three movies, five video games and numerous toys and models. [1] [2]

From this we can infer that adding such features to a production car has the potential to be well received by customers and that further integration with additional artificial intelligence (ex. voice assistants) is another topic worth exploring. Possibility for further market promotion of the vehicle, as well as popularization of science and education (namely computer engineering and AI/ML engineering) is also to be considered, although care must be taken not to violate the intellectual property of the original television series.

[3]Knight Industries Two Thousand

## 1.1 Touch Screen Interface Complexity Analysis

I have carried out a simple analysis of how many touch screen interactions are required to adjust various comfort settings on our car of interest (2020 model Škoda Kodiaq with MIB 2.5 radio and optional Canton sound system upgrade). In the below table 1.1 we see various "commands" that a user (passenger or driver) may want to issue to the car's comfort controls, and thus how many interactions can be conserved by the use of an intelligent control framework, which would then be capable of reducing the number of interactions down to one.

As some adjustments to the car's control function require both button presses and touch screen interactions, I have recorded button presses in the table separately. Furthermore, the amount of touch screen interactions for a single action can actually **vary** depending on the state the comfort controls have been left in by the previous interaction, and such state may be permanent (e.g. not time out and change back to a default state). This happens notably in the case of changing radio stations due to the paging of the radio station presets, and thus almost completely precludes the use of muscle memory, forcing the driver to break attention on the road and look at the radio.

An uncertain amount of touch screen interactions needed to carry out adjustments also happens in the case of sound volume adjustment, where a well-known volume knob has also been superseded by capacitive touch buttons. I have counted those interactions as touch screen interactions, as the buttons have the same tactile feeling as a flat touch screen and are located in close proximity to it.

| Action | Touch Interactions | Physical Button Presses |
|---|---|---|
| adjust sound volume | 1 to several | n/a |
| set sound source | 1 to 4 | n/a |
| adjust steering wheel heating | 1 | 1 |
| adjust seat heating | 0 to 1 | 1 to 3 |
| set independent heating | 3 to 5 | 0 to 1 |
| adjust rear seat temperature | 1 to several | 0 to 1 |
| set driving mode | 1 | 1 |
| set (adaptive) cruise control | n/a | 2 to several |
| set speed limiter | n/a | 3 to several |

Table 1.1: Required Touch Interactions for Various Controls of the Car's Comfort Functions.

Measurement taken by the author.

At this point it can be argued that the driver can control the comfort functions safely using the steering wheel controls, coupled with the modern multi-function instrument display, instead of focusing on the radio's touch controls. While this is possible, not all functions can be controlled in this manner (namely adaptive cruise control and air conditioning), and the issue

with breaking attention (looking away from the road) still persists, mainly in drivers that are inexperienced with the particular make and model of the car they are currently driving.

Figure 1.2: Example of an inexperienced driver behind the wheel of a test vehicle.

## 1.2 Implementing External Control in an Existing Production Car

Recording and analyzing messages on the CAN is easy, because it is a bus-type communication. This holds true as long as the definitions of the messages that come over the bus are known, since the messages themselves hold raw binary data with no descriptors. On the other hand, reliably introducing external data into a CAN network of a finished production car without significant changes, is a difficult task. Save for the vehicle's diagnostics, no part of the electronics is intended to accept commands from an external, previously unknown source.

The basic theory of how to accomplish this is as follows: Create a gateway that will intercept messages on the CAN bus of the vehicle and then modify their contents according to the desired changes. This is further complicated by the fact that some vehicles use additional, higher level protocols on top of the standard CAN bus communications, such as the Bedien und Anzeigeprotokoll (BAP) protocol in case of cars produced by Volkswagen AG.

This is not a trivial matter and it constitutes a topic for a thesis of its own. Luckily, this has already been described and solved by Ing. Jaroslav Beran in his thesis named *Firmware for Control Module of an Intelligent Vehicle*. [3] The resulting software and firmware from his thesis will be used to implement control functionality, together with hardware CAN gateways by the Department of Measurement. [4]

It needs to be stated that if such functionality is to be integrated in a future production vehicle straight from the factory, a different approach will likely be taken. External control functionality could be implemented through changes in the vehicle's infotainment or electronic control units.

## 1.3 Displaying Custom User Prompts on an Existing Production Infotainment Screen

The most pragmatic approach in adding custom features into a pre-existing vehicle is to just add a new device on the dashboard, somewhere within easy and convenient reach of the driver. This has been commonly seen in practice for as long as vehicles have been modified – examples being taximeters, citizen band radios, blue light and siren controls in rescue vehicles, or even something as trivial as smartphone charging mounts. This makes it easy to imagine that an AI/ML based driver aid could be controlled from a mounted tablet computer or something similar.

A more elegant solution, however, has been provided to me and my department by the car's manufacturer in the form of a custom display source switch with HDMI input. This is made possible as the infotainment system in the car in question is split into two parts – the display and the main unit. Those two parts are connected by an LVDS cable, which allows the inclusion of a display switch between them. The display switch can then switch the picture visible on the main infotainment screen for a custom one generated by my system.

As the touch information is relayed to the main unit via communcations on the CAN bus, a CAN gateway, [4] same as the ones used in Ing. Jaroslav Beran's thesis, can be used to intercept said touch information. This will have a twofold effect: Disable unwanted input to the vehicle's infotainment while the picture is swapped for our own prompts, **and** allow us to evaluate whether the user has touched our Yes or No button.

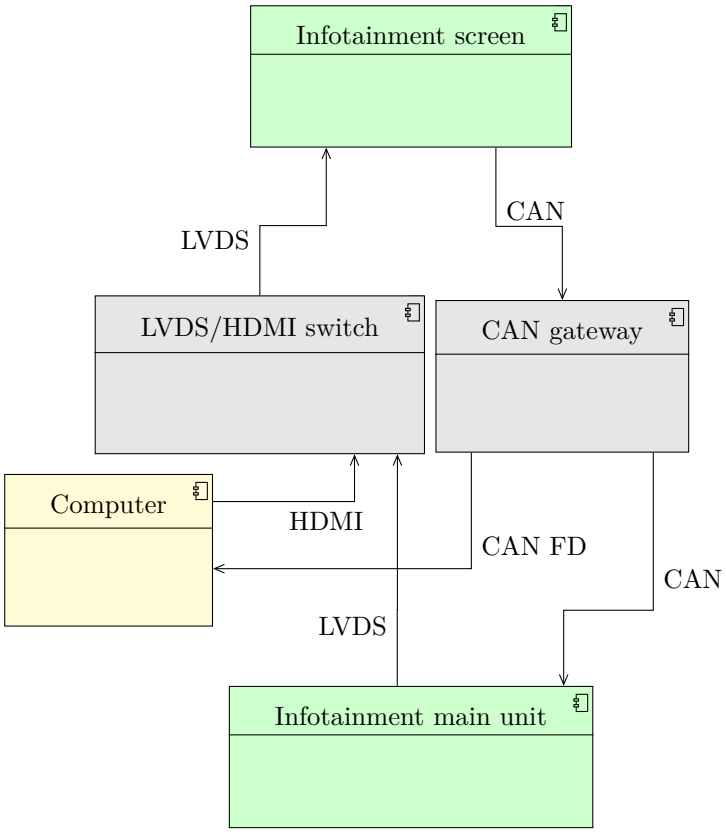


Figure 1.3: Component diagram of the approach to display custom prompts on the car's infotainment screen.

Components in light green are original parts of the car, and components in grey have been provided externally and are not by the author.

This is a good time to mention that the CAN gateway, made by the Department of measurement of my faculty, uses the more modern Controller Area Network Flexible Data-rate (CAN FD) protocol to communicate with the computer. That is because in some use-cases foreign to this thesis, there is a requirement for high bandwidth on the control interface. This changes little, aside from the need to select an appropriate CAN FD interface for the computer.

## 1.4 Hardware Platform for In-Car Computing

My supervisor has assigned me to use the Advantech ARK-3520P fanless embedded box computer, which was already in the Department's possession. I had no objections to this, as this kind of device seems to be perfect for such application because it has many good capabilities for use as an in-car computer

according to its datasheet [5]:

- Fanless design prevents long term dust accumulation.
- Wide operating temperature range (-20°C to +60°C).
- High vibration and shock tolerance (30 G shock, 3 G root-main-square vibration).
- DC power supply with wide operating voltage range (9 to 36 VDC).

However, as the computer sadly doesn't provide any CAN FD interfaces (neither built-in, nor as optional expansion), we needed to add an external interface. We have decided to use the Kvaser Hybrid Pro 2xCAN/LIN USB interface due to the Department's previous experience with it, good support on the Linux operating system, and a very wide operating temperature range.

Figure 1.4: Advantech ARK-3520P fanless embedded box computer. Image provided by the manufacturer.

## 1.5 Conclusion of the Introduction

Simply and shortly put, we are going to build a car that can control its own comfort functions (such as air conditioning or cruise control) using an AI/ML system. This is so that the driver can focus on the road and not be distracted. The idea is for the car to behave as if the driver always has a passenger with him / her, and for that imaginary passenger to be in charge of controlling the car's various comfort functions, reacting to events in and around the car on its own. The driver is only asked to confirm or refuse the action.

Chapter 2

# Top-level Architecture of the Framework

Having spoken repeatedly about artificial intelligence and machine learning, I turn to my study results report, and with great shock realize that I have not studied a single subject nor had a single exam on this topic, aside from some basic statistics. This makes sense, as I am studying Computer Engineering, a specialisation that does not deal with such matters. Thus it is clear that the artificial intelligence will need to be separate and also modular from the framework, in order to be supplied by other parties.

Keeping this in mind, and when we forgo the motivation talks, questions of popular culture, business marketing, and user convenience, all discussed in the previous chapter, we are left with the actual engineering task laid before us:

"Give the car a REST API."

The basic idea is as follows: The artificial intelligence models will reside on their own separate server, or even multiple servers. They will communicate with the car API over the HTTP(S) protocol, following the REST architectural style. Each model will have its own endpoint in the API, and it will be able to fetch its relevant data (values gathered from the car's systems) at its own pace via GET requests. When the model decides that an action is to be taken, it will issue a POST request with the relevant details in it. This action request will be further parsed and validated by the server running the car API.

## 2.1 Comparison to Existing Solutions

Considering all the specifics I spoke about in the above paragraphs (and in the introductory chapter), a parallel can be drawn between the outlined system

and some readily available message queuing systems (also known as message brokers). Examples of those include Apache Kafka, RabbitMQ, RocketMQ or Apache Pulsar. Such systems are used in large scale projects with long-term operation or for a large amount of data being exchanged.

A message broker enables different applications (or application components) to communicate via passed messages, which can be individually collected on-demand or directly streamed. The messages can be separated into different topics, and can even have multiple publishers (senders) and subscribers (receivers) for a single topic – the capabilities of such systems are vast. [6]

While a message broker system is capable of accomplishing the task of collecting data and serving them to AI/ML models, using it to control just a single vehicle would be excessive. Implementing adapters to reliably pass data into such a queue, and to extract it back again, would take as much consideration and careful design as implementing a data broker from scratch.

If such a system for control of an intelligent vehicle is deemed marketable and effective, a large-scale, horizontally scalable message broker, such as Apache Kafka, may be implemented for passing messages from an entire fleet of vehicles.

## 2.2 Outside-Facing API

The tasks required to be carried out by the outside-facing "data broker" that the AI/ML models will interface with are as follows:

- Provide an API for each individual registered model.
- Validate actions issued by registered models.
- Allow registration, modification, and deletion of individual model API endpoints (CRUD configuration capability).
- Keep track of different drivers, allow CRUD configuration capability for them, and report the current driver to registered models.
- Optionally allow for different data formatting in actions issued by registered models (as they will be provided by different suppliers).
- Optionally persistently cache collected signals in case of system restart.

As we assume there will be a multitude of various AI/ML models requesting their data at various paces (slow to very fast), a robust and effective web framework for constructing the car-side server must be chosen. I have decided to use the C# programming language due to my previous experience with it, and the ASP.NET Core framework from Microsoft, as it is presently

considered to be one of the fastest web frameworks. For example, it consistently reaches top 10 in the TechEmpower Web Framework Benchmarks in overall (composite) score over the past years. [7] Other high-ranking frameworks from this test were considered, but ultimately the choice stayed with ASP.NET Core, mainly due to the renown and long-standing support of the ASP platform by its creator Microsoft, and due to ease of development.

I decided to use SQLite as the internal database of the car-side server as there will be no extensive database functionality required; and the ORM functionalty provided in the ASP.NET Core framework by the Entity Framework Core component natively supports SQLite as its backend. The purpose of the database will be twofold: To store the configuration and registration data of individual models (and thus their API endpoints), and to provide persistent storage of "last-seen" signal values for fault tolerance. This will be explained in-depth in the coming chapters.

To enable widely **individual** processing and validation of data coming from the various AI/ML models, I took a relatively novel approach, which is to use the IronPython implementation of the Python programming language. This enables me to embed scripting functionality directly into the "data broker" C# server. As such, every registered AI/ML model will have its own configurable Python script, which will be executed every time data is submitted by said model. Thus it can be used both for validation (to protect against spurious model behaviour) and to enable it to handle any possible variation in data formatting between different models.

## 2.3 Inside-Facing Functionality

With the "data broker" role fulfilled, we have three more sub-tasks at hand:

- Data acquisition from the car's communication networks (DAQ).
- Communication with the vehicle's occupants (HMI).
- Commands to the car (Activator). (Solved by Ing. Jaroslav Beran.)

All of the above sub-tasks are, without doubt, hardware dependent, with the term "hardware" including the car itself. A different model or a different manufacturer will use different communication networks or have them set up differently, jeopardizing the compatibility of DAQ and Activator tasks. Most likely a different vehicle infotainment or "radio" will be used as well, meaning the HMI task will also be dissimilar. For this reason we can label these three sub-tasks as **car-dependent**.

In stark contrast, the REST API (data broker) is, in its conception and implementation, independent of the car's specifics, and with sufficiently modular design, could even be decoupled from the car itself in future updates to the framework. Thus, we can label it as **car-independent**.

To make a clear separation from an architectural point of view, and to enable decoupling the car-dependent and car-independent parts of the system in the future, a second, very simple REST API is proposed. It will be queried by the data broker and translate its commands to the DAQ, HMI and Activator parts of the system. This will also facilitate the "porting" or expansion of this thesis' result to other car models and manufacturers.

I have decided to create this unifying car-dependent API in the Python programming language and the Flask microframework for its ease of use, small size and simplicity of development. Flask is a minimal framework intended for the development of websites and APIs made by the Pallets Projects community. While this combination is considered to be among the slowest of web frameworks, placing somewhere around the **bottom 20** in the TechEmpower Web Framework Benchmarks [7], this is not significant, as the unifying car-dependent API will not be handling any data-intensive tasks.

## 2.4 Selection of Internal Communication Methods

Speaking of data-intensive tasks, the stream of collected data from the DAQ component will be constant. Although the present solution operates with only one connected CAN bus interface – meaning that the raw data throughput is not immense – we have to keep in mind that future versions can add more data inputs, for instance additional CAN bus interfaces or even different networks such as Automotive Ethernet (100Base-T1) – with the possibility of very high data rates.

This makes it clear that the DAQ component is going to need a direct, continuous, streaming connection to the upper-layer data broker. This connection will be made on-demand when the data broker starts up, and will skip the "unifying" Python API. In order to maintain the separation of car-dependent and car-independent parts of the system (to enable the eventual decoupling of them), a secure connection is needed.

I elected to use JSON streaming over a TCP socket, the latter being secured with SSL/TLS, for this task. JSON streaming (namely newline-delimited JSON or NDJSON for short) was used because I am already using the JSON format extensively in other parts of this thesis (namely in the REST APIs) and thus it offered easy integration with my existing tools and libraries. I considered my options between TCP, UDP and SCTP, for the socket protocol and ultimately ruled out UDP because of its lack of guarantees, and SCTP because of low support, leaving TCP as the clear choice. A more in-depth analysis of the differences between those protocols was carried out by Ing. Jaroslav Beran in his thesis.[3]

SSL/TLS is considered to be the de-facto standard for encryption in socket communicaton, and is widely supported by many free and open-source libraries for many programming languages. A streaming connection secured in this

manner will nicely compliment the use of the secured HTTPS protocol for transactional operations.

The last part left to decide is the method of communication between the car-dependent API and the individual modules that will be controlled by it. As I already had the Activator module provided to me, I took inspiration in its control method and decided to use the same for the HMI and DAQ modules to keep it simple.

The control method in question is JSON-RPC, a specification for remote procedure calls, which uses the JSON data format. [8] It is very simple to implement and use, especially with JSON being used elsewhere in the project. Utilizing it in conjunction with a TCP socket allows us to manually control and troubleshoot the modules using the `netcat` (or `nc` for short) command on Linux.

During the project's development it proved necessary to acquire more complex control messages from the car. These are encoded with the BAP protocol, and for decoding them I was supplied a Linux command-line utility from the car's manufacturer. Sadly I was not provided with the source code, which precluded me from importing the utility's functionality directly into the DAQ module as a library.

However, the utility can be configured for output of the decoded messages into the standard output, conveniently also in the NDJSON format. Thus, I decided to pipe its output into my DAQ module and implement decoding and re-encoding of the messages coming from said utility. Hence, they will be mixed on the output of the DAQ module with the messages that the module itself decodes directly from the CAN bus.

## 2.5 Overall Layout

The below figure 2.1 shows the whole layout of components and communication protocols that I have described so far. The "outside facing" HTTP(S) REST API that the Data broker / Middleware part will use to interface with external AI/ML modules is not pictured.

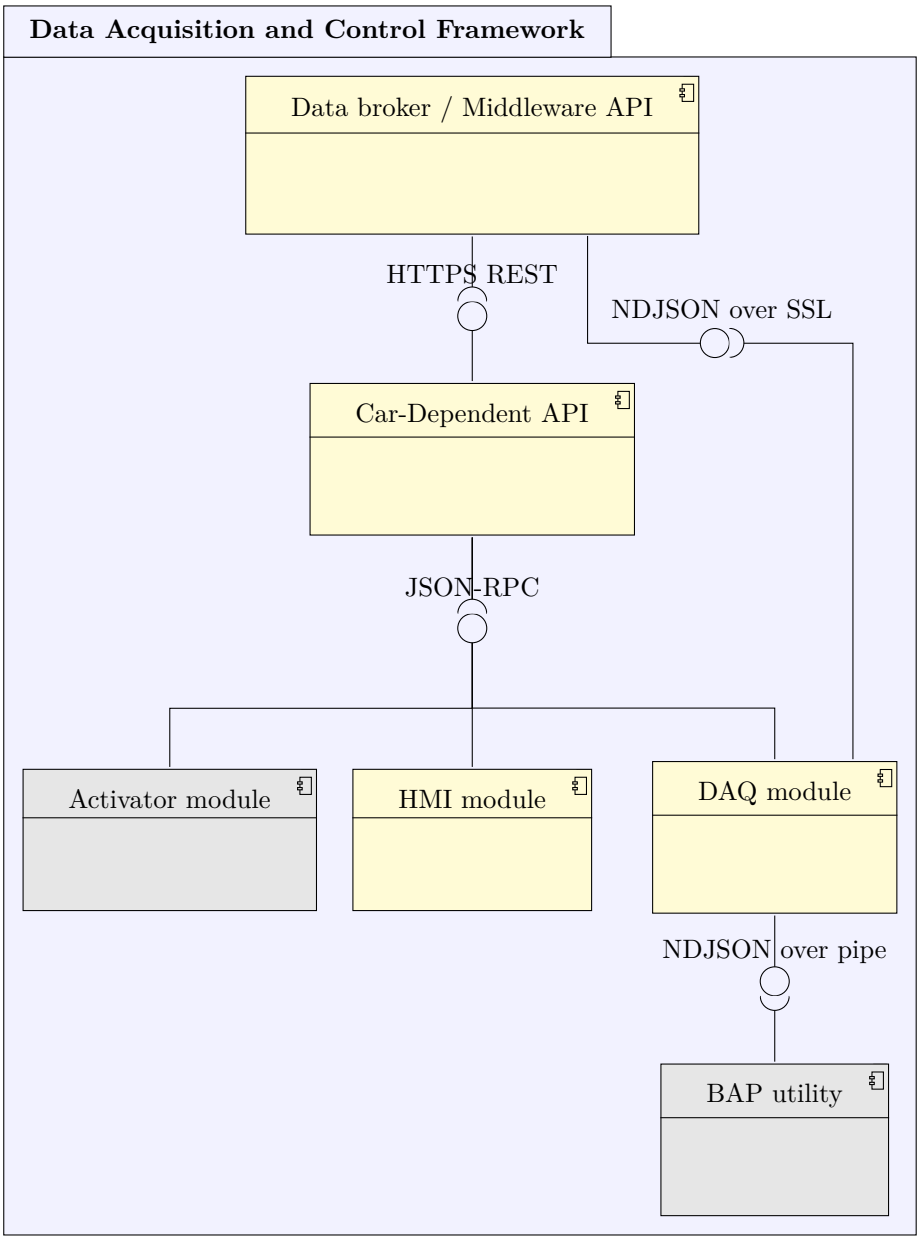


Figure 2.1: Component diagram of the system's top-level architecture.
Components in light gray have been provided externally and are not made by the author.

## 2.6 Bring-up Behaviour

My description of the whole system's layout and architecture has made one thing clear: It is far from trivial. The outward-facing data broker API depends on the functionality of the lower-level systems under it. The DAQ module needs to establish the SSL connection to the data broker API in order for the system to function properly.

We can roughly split the system bring-up into three parts:

- Operating system configuration (bring up interfaces, prepare log files).
- Car-Dependent API startup.
- Data Broker API startup.

Operating system configuration will be described in later chapters. After the operating system starts, the Car-Dependent API can then be started along with the Data Broker API. The latter API will poll the Car-Dependent API via a special heartbeat/health check endpoint. If this fails, the bring-up of the Data Broker is paused and the polling is retried after a few seconds. This means the bring-up is quite simple from the point of view of the operating system, as pictured in below figure 2.2.

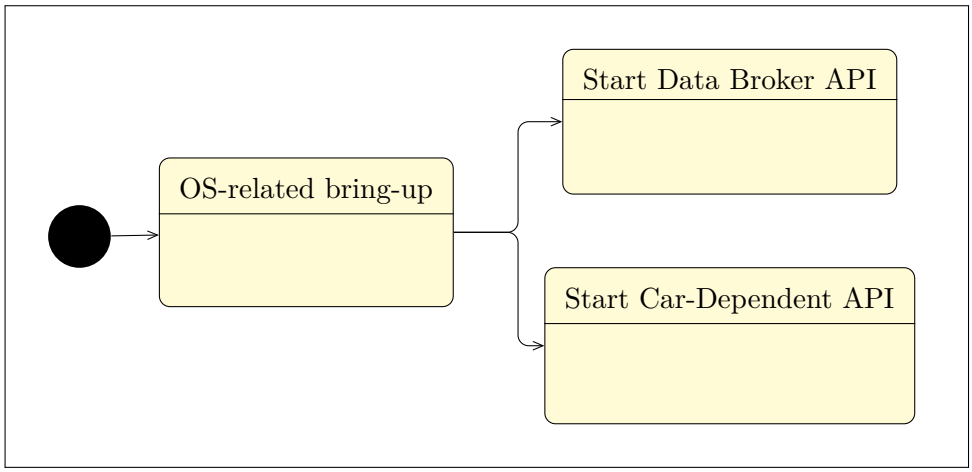


Figure 2.2: State diagram of the overall bring-up behaviour.

As the Car-Dependent API is going to be run on the same hardware the actual modules are running on, it can be tasked with bringing them up and monitoring their health status. This is facilitated by the rich functionality of the Python programming language's libraries, namely the `subprocess` module. Figure 2.3 shows this sequence.

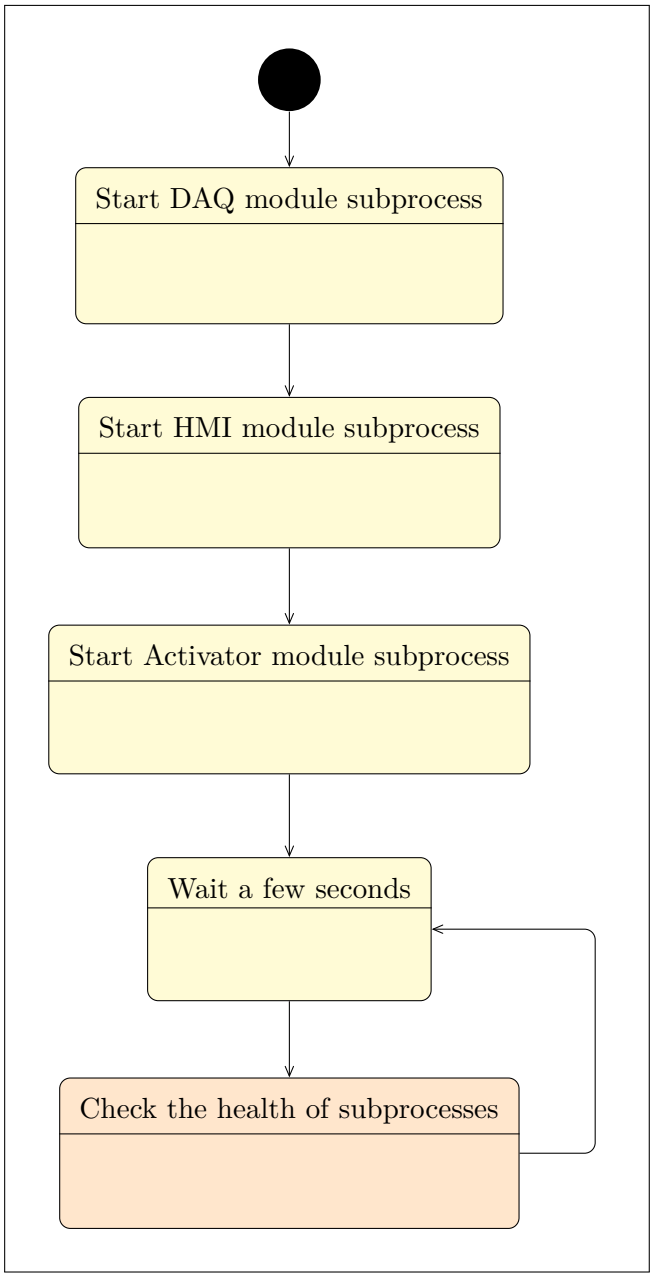


Figure 2.3: State diagram of the Car-Dependent API bring-up behaviour.
The "Check the health of subprocesses" state can be considered final, but due to repeated waiting and re-checking I decided to not label it so.

Only after the Car-Dependent API is established to be working and ready, will the Data Broker continue starting up and then issue a command to configure the DAQ module connection to itself. Thus, the "circle" visible in figure 2.1 consisting of HTTP(S), JSON-RPC and NDJSON over SSL is completed.

The figure 2.4 beneath illustrates this wait-and-retry mechanism and subsequent configuration upload to the Car-Dependent API.

For debugging purposes, I later added a feature that allows us to turn this behaviour off via a configuration file belonging to the Data Broker API, so that its bring-up will not be paused. This means the Data Broker API only serves old signals previously persisted[4] to its database, or none at all if the database is empty, so this functionality should always be preserved in production use.

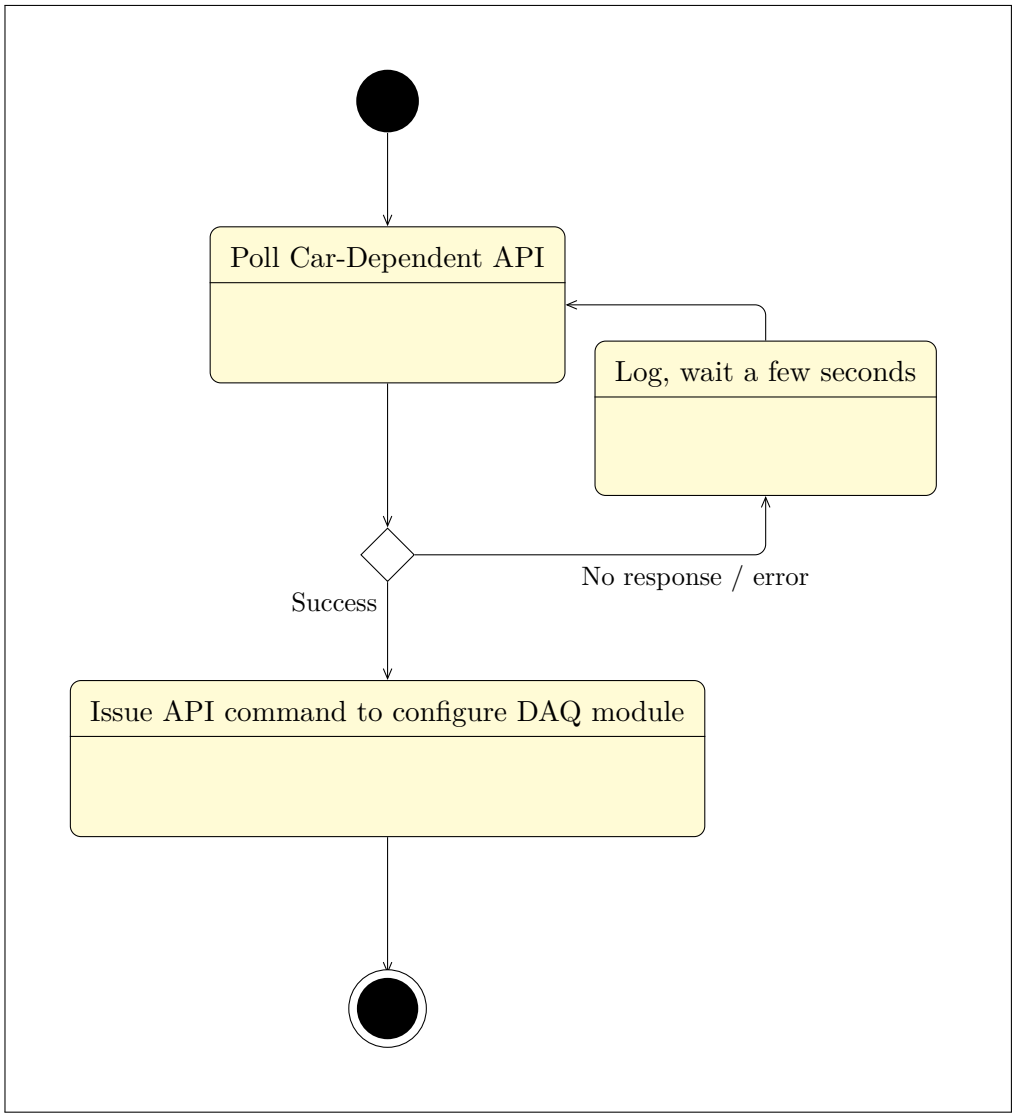


Figure 2.4: State diagram of the Data Broker API bring-up behaviour.

[4]Data that has been saved to the database during a previous session, together with appropriate timestamps.

Chapter 3

# Design of Individual Framework Components

All of the framework components will need to leverage asynchronous programming techniques – the exact implementation of them is going to depend on the used programming language and/or framework. In the case of the DAQ module, it is required to handle multiple data sources (CAN data coming in and also piped data from the BAP utility) together with interprocess communication via JSON-RPC. In the case of the HMI module, it will need to handle drawing the UI, CAN data coming in from the touch screen, as well as interprocess communication. The Car-Dependent API has to schedule regular health checks of its child processes. Finally the Data Broker API must asynchronously accept data collected from the BAP utility, and also schedule persisting collected signals to its database for reliability.

## 3.1 Data Acquisition Module

The job of the Data Acquisition (DAQ) module is, as its name suggests, to acquire raw data from the car's communication networks, and to decode it so the Data Broker can make the individual signals and their values available to the AI/ML models that interface to it.

Originally, the DAQ module was only intended to capture unstructured communication on the CAN bus, using the so-called DataBase CAN (DBC) files. Before I explain what the function of the DBC files is, let's first remind ourselves how the CAN bus works from a programmer's perspective. When the CAN interface is correctly set up and the communication is established, we start receiving a stream of CAN frames (also called messages). These frames contain nothing more than an identification number (ID) and one to eight bytes of binary data. How do we make sense of this?

The DBC files contain descriptions of the various signal values that are

transmitted in said CAN frames. Those descriptions include a human-readable name, a unit of measure (if any), and in which frame the signal is contained (by frame ID), and which parts of the frame's data correspond to its value.

These make DBC files a considerable benefit in accomplishing our task. Nonetheless, we still have to look at it from the programmer's perspective: How do we parse and utilize these files? There exist a few free, open-source libraries for dealing with such files, however, I ran into multiple issues when using them with DBC files provided to me by my supervisor.

This part of the story actually began one semester before the time of writing this thesis. The Department of Measurement has its own internal DBC parsing library, and my supervisor has assigned me to use it. The library worked flawlessly with any files that I could test it out with, however, it was Windows-dependent and built with Visual Studio. Because of this dependency, my semestral task back then became the cleaning up of this library (from this point known as DbcFileC) and porting it to Linux.

### 3.1.1 Porting Legacy C++ code from Windows to Linux

The original DbcFileC library, as I received it, dated from 2003 and contained approximately 5000 lines of code. It combined seven C++ classes in one source code file and one header file. As it was originally built with an old version of Microsoft Visual Studio, I struggled to compile it on my work computer, which had a newer version of it. Clearly, there was considerable housekeeping that had to be done.

My first goal was, obviously, the successful compilation of the entire source code. As simply transferring the code to a new Visual Studio project resulted in arcane linker[5] errors, I decided to recreate the project as a CMake project. This had a twofold benefit. First, the CMake toolkit facilitates easy compilation on different platforms (as opposed to Visual Studio projects). Secondly, it immediately compiled the source code (still Windows-only at this point) without any issues. Since Microsoft Visual Studio itself natively supports CMake integration from the 2017 version onward[9], there were no further obstacles for me to work around.

Next, I worked on separating the library's classes into separate source and header files, and updated the CMake configuration accordingly. As the library was using the Boost library for its serialization functions, I updated the used Boost library version to a more recent one. Because of the pre-existing usage of said library, I also created unit tests using its `Boost.Test` component in order to verify that I had not broken anything while making changes to the source code.

At the same time, I worked on transferring the information and notes first found in the many comments located in the source code, into a form supported

[5]The linker is a program which is a part of the compilation toolchain. It combines individual compiled object files into a final executable file in the last step of the compilation process.

by the Doxygen documentation generator tool. I did individual tests and consulted my supervisor in order to document the parts of the source code that weren't previously clear in their function. This resulted in the ability to generate an all-inclusive documentation PDF for the newly remade library.

With the order restored in the file structure, documentation available, and unit tests ready, I commenced porting the library to Linux. The library exclusively used Safe String Functions[6], which were easy to change back to their standard versions. Also, some odd Win32 functions were used to make the past programmer's job easier, and those were also simple to remove. The largest issue was in the library's use of wide characters[7] and wide strings[8]. This was done, no doubt, in order to enable the use of accented and other non-standard characters in the DBC files, as the files may come from the German manufacturer and may contain, for example, umlauts (¨) or Eszetts (ß).

Windows facilitates the use of international characters using the UTF-16 encoding, namely UTF-16LE (Little Endian). [10] [11] Its internal API functions thus accept the aforementioned wide characters (C/C++ type `wchar_t`) and subsequently wide strings (C++ type `wstring`). Those two mentioned types are thus assumed to be 16 bits (2 bytes) long.

Linux (and Apple macOS), however, use the UTF-8 encoding for international characters. While the previously mentioned UTF-16 encoding uses more space per character to enable a wider array of possible characters, UTF-8 uses a coding sequence of variable length for each character that ranges from 1 to 4 bytes long. [12] Its encoding table is laid out in such a way that the most commonly used characters match ASCII at a length of 1 byte, and the sequence gets longer the rarer the character is. This means that any text, international or not, is composed of individual parts that are always 8 bits (1 byte) long.[9]

This signifies that the Linux API functions accept regular-width characters (C/C++ type `char`) for both English-only and international texts. The type `wchar_t` has a **different meaning** on Linux (and macOS), being 4 bytes (32 bits) in length instead of 2 bytes (16 bits) on Windows. As it's not used in the Linux APIs, the occurence of `wchar_t` in programs on this platform is very low.

To summarize, the wide characters are a requirement on Windows, while they are almost unused on Linux. Furthermore, source code using them is not portable, as they produce different data widths, and use of them in standard

[6] Non-standard extension to standard library functions such as `strcat`, `strcpy` or `sprintf` by Microsoft.

[7] Wide characters are characters represented by a wider space in memory (2 bytes or more) instead of the usual single byte used for normal characters.

[8] Wide strings are strings composed of wide characters.

[9] As a consequence, a text with accents and other non-standard characters just takes up more space, compared to a text with no accents.

library functions (namely string formatting) may not be equivalent between different operating systems or compilers.

Now, how do we solve this? While researching this problem, I came across a manifesto called *UTF-8 Everywhere* [13], which calls for the ubiquitous usage of the UTF-8 encoding and provides many excellent reasons to support this choice.

At this point, I received another benefit of upgrading the included Boost library version at the start of this library's re-development. Boost has recently gained a new component, called `Boost.Nowide`. This wrapper library easily and conveniently wraps[10] standard library functions on Windows and makes them UTF-8 compatible, while leaving the original UTF-8 compatible functions on other platforms unchanged. This, as a result, allows me to convert the whole library from the use of wide characters and wide strings to regular characters and strings – all while preserving the capability to process accented and other non-standard characters. With this, we have **finally** gained cross-platform compatibility.

Lastly, since some DBC files arrive from other parties encoded in the CP1252 encoding, I added a simple component that enables the library to correctly parse this encoding into UTF-8. As a result, the library now solely uses the UTF-8 character encoding in its internals.

At long last, the library was ready for use on Linux and I created a simple example program which uses the SocketCAN component of Linux to communicate with a CAN interface, and which demonstrates the successful parsing of received CAN frames.

### 3.1.2 Efficient Implementation of Data Processing

As was mentioned at the very start of this chapter, the DAQ module has to continuously process the stream of data coming in from the CAN interface. At the same time, it has to be responsive to configuration requests (JSON-RPC) incoming via a TCP socket. This implies that we need to use a threaded or event-driven approach to accomplish this task.

I resolved to use the event-driven approach, mainly for reasons of reliability and simplicity (with no need to be wary of data integrity). Because of previous knowledge and a wide array of provided functions, I decided to use the Qt library to build the DAQ module, despite it not being a GUI application. The library provides the event loop itself, as well as functionality for communication over the CAN bus, SSL encryption for TCP sockets, and JSON parsing functionality.

The basis of the module's operation is quite simple. After starting up, it initializes its CAN interface and TCP server for receiving JSON-RPC calls.

[10] This means to enclose an existing API interface using a minimal layer of new source code in order to refine it or make it safer to use.

In the below figure 3.1 the module's way of handling said JSON-RPC calls is described.

Handling an incoming connection (`Daq::handleTcpConnection()`) and storing its data (`Daq::readTcpData()`) are separate procedures, with the second one actually also being triggered by an event. This is to handle processing partially received data – in other words, to not have the requirement for the JSON data to be received all on one call to `QTcpSocket::readAll()`. This may happen due to the processing speed of our computer and also due to the nature of the TCP communication.

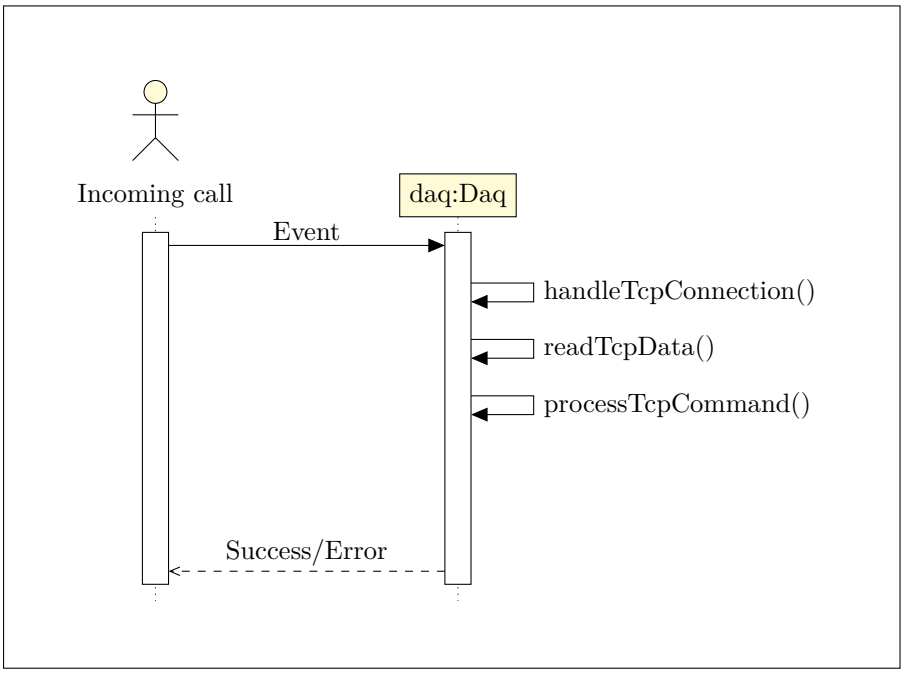


Figure 3.1: Sequence diagram of the Data Acquisition module's RPC handling behaviour.

The main function of the JSON-RPC commands is to configure the DAQ module as to where to send the parsed data – what the exact port and IP address of the already running Data Broker component is. If no such data client is connected, parsing of CAN frames will not occur, as it would be pointless.

Finally, after all checks are passed and the (`Daq::readTcpData()`) function senses the termination of the JSON-RPC command by a NULL character, the command is processed. As the reference to the original JSON-RPC client is passed through the call hierarchy, command success or error is reported to the JSON-RPC client.

The processing of incoming CAN frames is also event driven, thanks to the seamless implementation of the CAN protocol in the Qt library's Serial Bus API. As shown in the below figure 3.2, it is even simpler than the handling

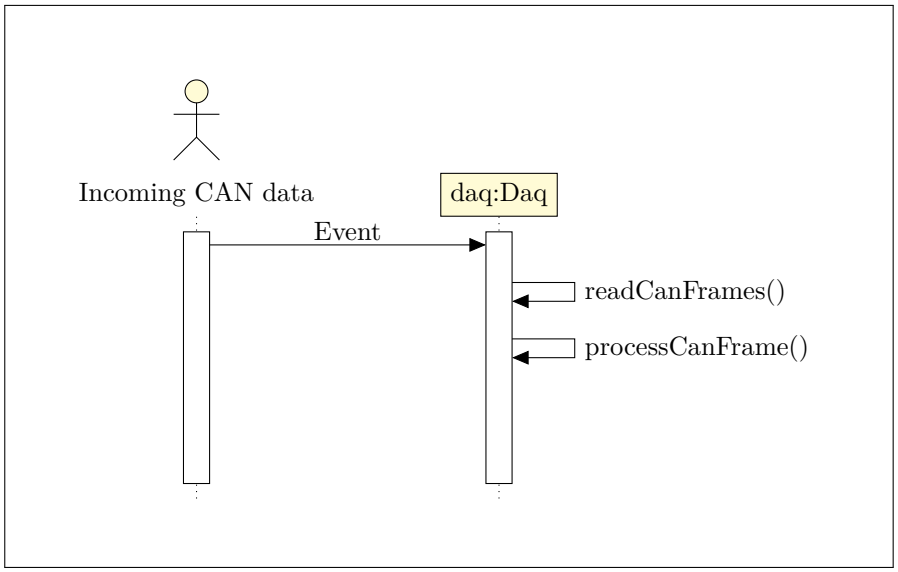


Figure 3.2: Sequence diagram of the Data Acquisition module's CAN data handling behaviour.

of JSON-RPC calls described above, as there is no return of data back to the CAN bus. Decoding via the aforementioned DbcFileC library to a connected data client is done in the `Daq::processCanFrame()` call.

### 3.1.3 Extending the Module for Acquisition of Structured Data

As mentioned in the introductory chapter, an additional request was raised later in the development of the system. It was to implement the decoding of more complex messages being transmitted over the CAN bus. This is done using the Bedien und Anzeigeprotokoll (BAP). Sadly, the protocol is proprietary and there is not much information freely available about it. What I know is that the protocol acts similarly to higher level protocols that we know and love from computer networks – that is, it uses multiple CAN frames to transmit a longer message.

Those frames that contain parts of BAP messages are not recognized by our simpler decoding approach due to having unknown (to us) IDs. I was provided with a proprietary command-line utility from the manufacturer. No source code was provided, however, the utility can be configured to output the data it decoded to the standard output, conveniently in NDJSON format.

I once again leveraged the functionality of the Qt library, this time with the QProcess class, which allows the programmer to easily start external programs and to communicate with their standard inputs and outputs in a reliable, event-driven way. As can be seen in the below figure 3.3, the parsing of the

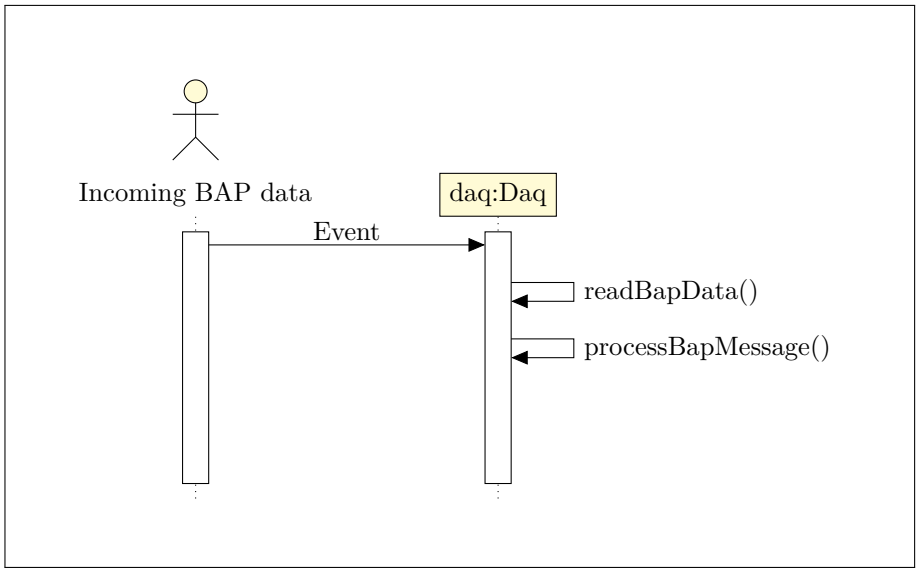


Figure 3.3: Sequence diagram of the Data Acquisition module's BAP data handling behaviour.

pre-decoded BAP data from the utility is otherwise very similar to the way the normal CAN bus data (found in single frames) is handled.

## 3.2 Human-Machine Interface Module

The Human-Machine Interface (HMI) module accomplishes the task of interfacing with the user, in our case the driver or the front passenger. Due to the uncommon hardware architecture of our system, as was first mentioned in the introductory chapter, the HMI module has an unusual amount of technologies that it interfaces to. Aside from the expected drawing of Graphical User Interface (GUI) on-screen, and communication with other components of our system (via JSON-RPC), it also communicates over the CAN bus and even uses a serial port. The reasons for this will be explained in detail in the following sections.

The HMI module uses the same screen as the car's infotainment system, which is enabled by the display switch supplied and installed in the car by the car's manufacturer. Aside from a regular HDMI display input, the display switch also requires a single low-voltage signal to select between the original infotainment picture and said external input.

Here, we run into a slight issue regarding what interface, on the actual computer, to use in order to drive said signal. All we need is a single programmable output on the computer, one we could call General Purpose Output (GPO). Although the Advantech computer has a connector called 'Digital I/O', containing 16 pins configurable for both input and output (GPIO), the

use of this connector requires the manufacturer's SUSI 4.0 API, which is not easily obtainable for Linux. [14] More of my troubles with the manufacturer's Linux support will be described in chapter 5.

A quick and simple solution for this was fashioned by my supervisor's colleague Ing. Jan Sobotka, PhD., which is to use one of the computer's many serial ports. The control signals of an RS-232 port can be fashioned as simple programmable inputs or outputs, although the particular direction can not be changed (the RTS and DTR pins are always outputs, CTS and DSR pins are always inputs). [15] Any differences in voltage levels are easily resolved with the use optocouplers or similar electrical components.

Referring to the first chapter again, namely section 1.3, the HMI module also does not receive user input like a normal GUI application would, and instead communicates over a CAN FD interface with an external gateway. Said gateway is used to divert CAN data containing touch events that normally go from the infotainment screen to the infotainment main unit. Now, instead, the data will be received on the aforementioned CAN FD interface. This means that the module has to control the CAN gateway, and decode the touch information from the received CAN frames.

### 3.2.1 Designing a GUI Program with Dual Functionality

I have once again selected the Qt library to build the module. In this case, its most obvious function is the creation of a Graphical User Interface (GUI) for the driver or the passenger to see. However, as was mentioned in the previous section about the DAQ module, it has many more functions regarding networking, JSON and the CAN bus, which were all utilized.

As the title of this subsection suggests, the functionality required of the HMI module is twofold. The first one, most notably, is to show prompts that offer the vehicle's occupants an action that a given AI/ML model has suggested, and allow them to accept or refuse it. (Example: "Would you like to set the Air Conditioning temperature to 25°C?") The second one, is to allow for selection of individual driver profiles when the car is started up. This is to facilitate the learning process of said AI/ML models, and also enable individual preferences (e.g. different drivers will probably want to set the cruise control differently on different occasions, and so on).

Switching between these two modes will be entirely triggered by a JSON-RPC call, and is not to be affected by the user in any way: The user is either selecting a driver profile or responding to an AI/ML model action. In order to accomplish this task, I used the QStackedLayout class from the Qt library. This class allows me to stack components of the GUI on top of each other so that only one is visible at a time. The ordering of this "stack" can then be changed programmatically. As long as this occurs before the car's infotainment screen is switched to the computer's output (and thus before the

HMI interface shows), the change between those two functions will be seamless to the car's occupants.

### 3.2.2 Connecting a Graphical User Interface to a CAN Network

Connecting such dissimilar things as a CAN bus interface and a GUI, seems like a daunting task. Luckily, using the Qt library's event system, together with its previously mentioned native support for the CAN bus, makes this easy to accomplish. Compared to the previously described DAQ module (which only received CAN data), the HMI has to send and receive CAN data in order to tell the CAN gateway to start (and stop) diverting touch data to it. Two more things must happen: The HMI module has to accept control commands over JSON-RPC calls, and must also use the RTS pin of a serial port to switch the infotainment display source. The combination of all these requirements causes the call sequence of an interaction with the HMI module to be considerably more complicated.

It should be mentioned that the below figure 3.4 applies only in the case of an AI/ML model action being presented to the user. Nonetheless, the only difference in case of the driver selection dialog being displayed is a class named `DriverSelect` being called instead of `ModelAction`.

An interesting point from the diagram to mention is the call to a function `MainWindow::acquireScreen()`. As there is only one screen connected to the HMI module and one interaction can take several seconds, care must be taken that a new command is not accepted if it arrives too soon after the previous one. Hence, the module must display a simple mutex[11]-like functionality in regards to external access. This will prevent concurrency errors in case of a mistake or misconfiguration of the system.

Only if the call to the function `MainWindow::acquireScreen()` succeeds, will the CAN gateways get configured for diverting touch data to the computer (function `GatewayControl::startCanIntercept()`), and the car's screen is switched to the computer's output (function `ext_hdmi_input_on()`). After the interaction with the user is finished, the car's infotainment functionality is restored by undoing changes that the mentioned functions caused (lowest three calls in above figure 3.4). For troubleshooting purposes, I also added a JSON-RPC call that can be used to request whether or not the screen is occupied, and thus if a new (module action or driver select) interaction can be initiated.

Last, but certainly not least, I need to mention that the HMI module implements a timeout on both possible interactions with the vehicle's occupants. The duration of this timeout is set to 5 seconds in the case of an AI/ML model action, and 15 seconds in the case of a driver profile selection. This timeout is

[11] Mutual exclusion, a programming construct used in concurrent programming.

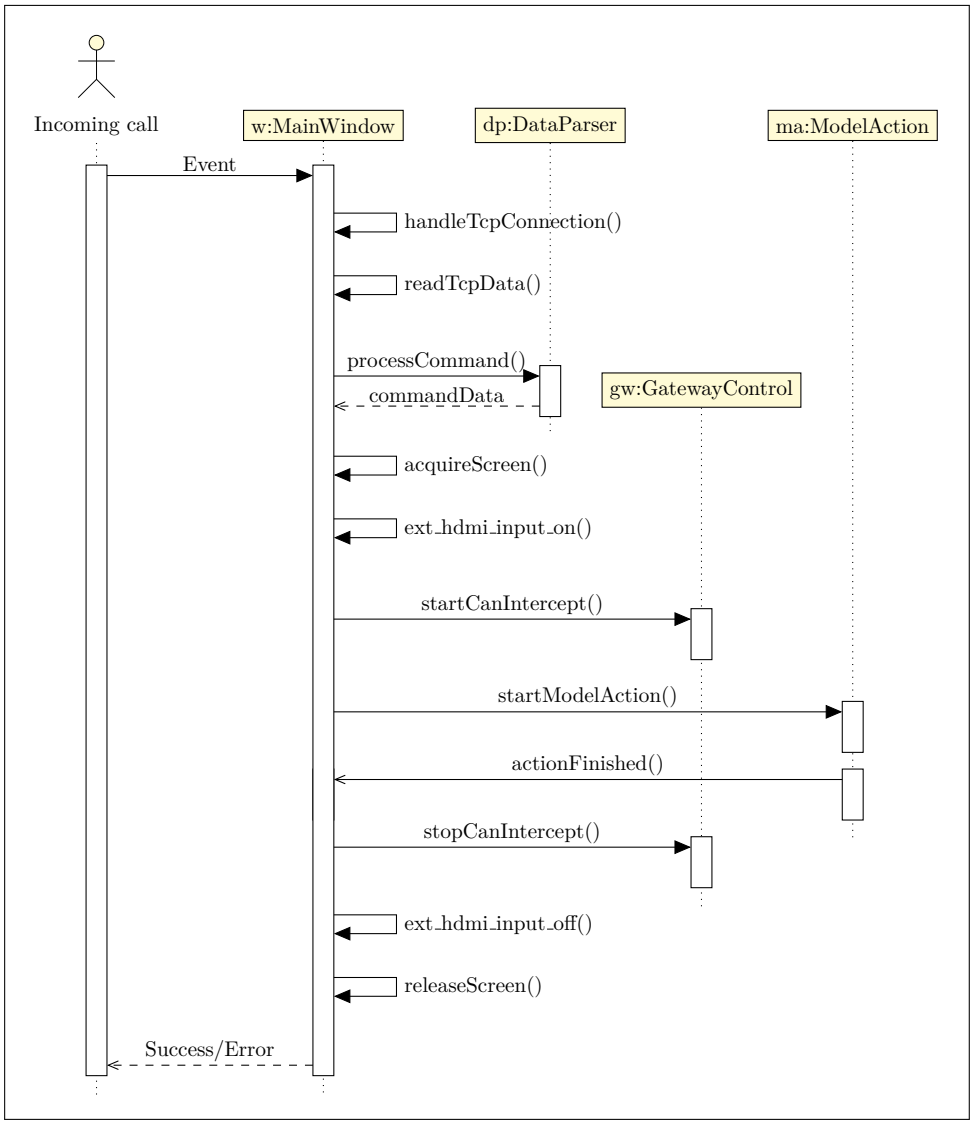


Figure 3.4: Sequence diagram of the Human Machine Interface module's RPC handling behaviour.

visualized to the vehicle's occupants by a progress bar running down. When the countdown reaches zero, with no interaction by the user, the result of the interaction is considered to be a refusal (”No”) in the case of a model action, and considered as the first, default, driver profile in the case of driver profile selection. A consequence of this is that the caller of a JSON-RPC request used to trigger one of mentioned interactions, has to wait for up to 15 seconds for a result.

### 3.2.3 Alternative Input Handling in a Qt Application

Previously, I mentioned that the HMI module will receive user input through CAN data that normally goes from the infotainment screen to the infotainment main unit, which will be diverted by the use of a CAN gateway when the module is active. This also means that there are no "regular" input devices (like keyboard, mouse, or touch screen) connected to the computer. As a result, the regular functions of the Qt library that handle user input have to be substituted.

Thankfully, the task to accomplish here is not very involved, as the only type of element (widget) that will need to respond to user interaction is a button, represented in the Qt library by the QPushButton class. This class has a property, called 'checkable'. Setting it to true makes the button behave similarly to a check box: Once pressed, it stays down. Afterwards the button can be programmatically depressed or released using the `QPushButton::setChecked()` function.

Described in figure 3.5 below, is the mechanism used for decoding touch information from incoming CAN data. Such events occur only during an active interaction of the user with the HMI module, as mentioned previously. Once again, this applies to an AI/ML model action being presented to the user, and, in the case of driver selection dialog, a class named `DriverSelect` receives the touch events instead of `ModelAction`.

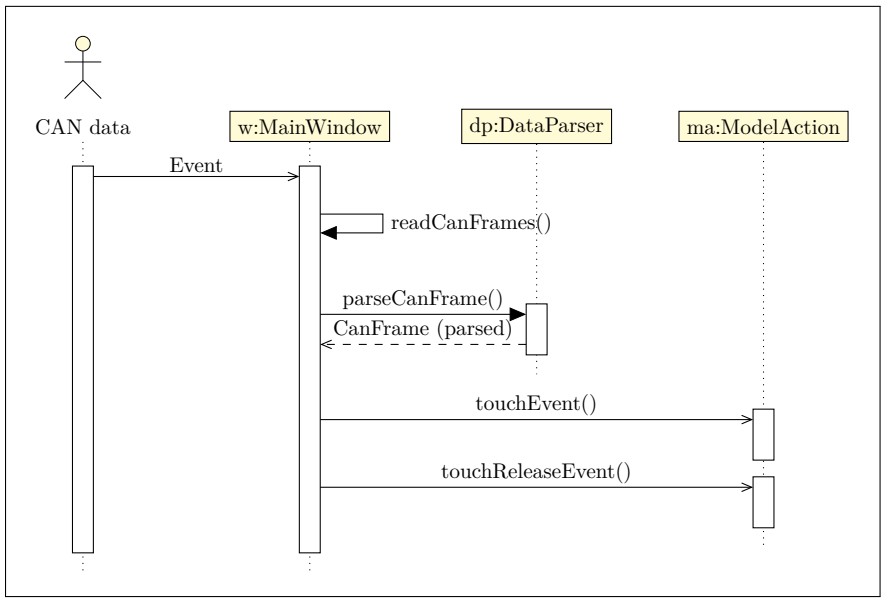


Figure 3.5: Sequence diagram of the Human Machine Interface module's CAN data handling behaviour.

As the user interaction data will be handled directly by the HMI module, we must also remember to offer the user a similar amount of responsiveness

that is known from touch-screen interfaces of today – be it in-car infotainment, or, for example, smartphones and tablets. That is: In order for the interface to be perceiveably "smooth" to the user, his choice must only be recorded after he or she releases the finger from the touch screen, not immediately on finger touch down.

If the choice was recorded immediately after touching the screen, most users would likely perceive the interface as broken or poorly done. This interface behaviour also allows the user to change or cancel a decision after touching the screen, by keeping their finger pressed down and dragging it to a different button, or aside from any buttons, and only then releasing the touch. The buttons should then react accordingly for a finger being dragged *over* them, that is, depress and release, but only stay depressed if a finger is released on said button.

Care must also be taken that all buttons are reset after an interaction is finished, so if the same kind of interaction occurs later, the user is not presented with a button that has already been pressed down.

### 3.2.4 Styling and Other Implementation Particularities

When the HDMI/LVDS switch is triggered, the computer's display output gets displayed on the car's infotainment screen whole. That means the picture will include any desktop environment elements, such as task bar, dock, or system notifications. While this matter surely will be intriguing to any occupants of the car who might have an interest in embedded computing or information technology in general, it would not be acceptable in production use.

Fortunately, the Qt library resolves this issue once again, because of its native support for use in embedded Linux environments. This allows us to completely forgo the desktop environment and even the windowing system of the Linux operating system, by configuring our program (the HMI module) to write directly to the computer's framebuffer via the `linuxfb` plugin of the Qt library.

Leveraging said plugin actually requires **zero** extra effort as it can be enabled on regular Qt programs simply by adding `-platform linuxfb` to the program's command line arguments, and making sure the program is launched with permissions to access the computer's framebuffer. This, however, raises a slight issue, in that the program running in such an environment has very plain visual styling compared to the same program running inside a desktop environment.

As seen below, I mitigated this by styling (decorating) the HMI module's GUI myself, using the Qt library's stylesheet functionality. I selected colors similar to the car's actual infotainment, so that it's not jarring to the vehicle's occupants when the HMI module prompt pops up. I also added a logo of my university, together with the name of our department, and a less visible text

on the lower right corner to remind the system's users that the decisions are powered by artificial intelligence.

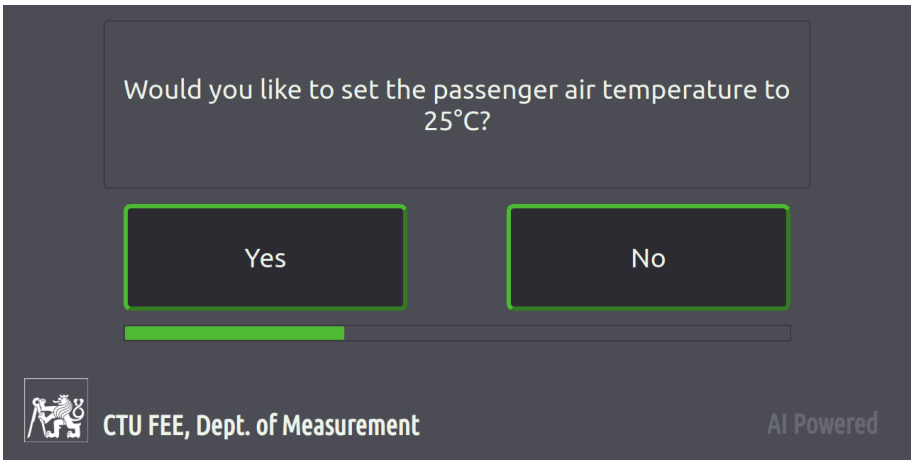


Figure 3.6: Screenshot of the HMI module's output during an AI/ML model action interaction.

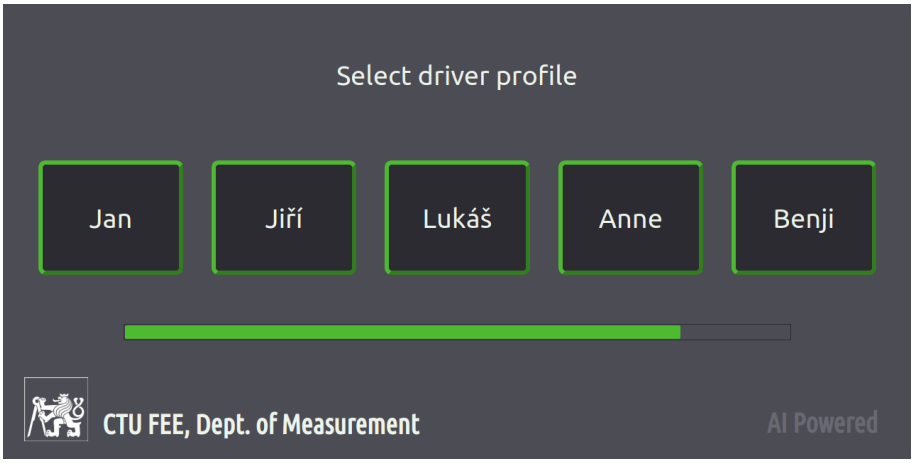


Figure 3.7: Screenshot of the HMI module's output during a driver selection interaction.

## 3.3 Car-Dependent API

The job of the Car-Dependent API is, simply put, to translate the JSON-RPC calls used to communicate with the lower-levels of our framework, into an easy to use HTTP(S) REST API, so that the upper-level Data Broker API does not need to support JSON-RPC by itself. The motivation for this is to make the Data Broker API easy to decouple from the rest of the framework, meaning it can be moved to a cloud server in the future.

Additionally, the Car-Dependent API is also tasked with starting the hardware dependent modules and monitoring their health, as well as with managing their logging facilities.

The API itself is written in the Python programming language and utilizes the `subprocess` module that belongs to the Python Standard Library. Additionally, it uses the APScheduler library for scheduling health checks on the subprocesses, and the Flask microframework to create its HTTP(S) REST API.

### 3.3.1 Checking Health of Subprocesses

All of the low-level modules, including Ing. Jaroslav Beran's Activator, support a heartbeat[12] JSON-RPC command. Thus, the Car-Dependent API is scheduled to issue this command by itself, every few seconds, to each of the underlying modules, with a short timeout period. If the heartbeat command times out and does not return a valid result, the module, being run as a subprocess of the API, is terminated and restarted.

The only issue with this approach is the DAQ module, which needs to be configured by the upper-level Data Broker API in order to stream collected data to it. This is resolved by the Car-Dependent API recording the values when the initial configuration command from the upper-level API passes through it, and replaying it to the DAQ module in case it is restarted.

Other than that, the API is very simple, being just under 300 lines of Python code in length, there is little to describe.

## 3.4 Data Broker / Middleware API

After we have spoken about every single underlying part of the whole system that this thesis describes, now is the time to finally talk about the top-level data broker / middleware API. Arguably, it can be considered the most important part of the whole system, as it is the part that the external AI/ML models will talk to from the outside. Not to forget, it is also the main talking point of the guidelines in this thesis' formal assignment.

As the Data Broker API is going to be the sole component storing a considerable amount of information (such as cached signal values, AI/ML model registration data and driver registration data) a more performant foundation to build it on is required. As mentioned in the previous chapter, I chose to use the ASP.NET Core framework, based on the C# programming language.

In order to facilitate testing of individual classes and components inside this API, I used the Dependency Injection (DI) technique when creating the Data Broker API. This technique, native to the .NET framework, replaces

[12] A heartbeat is a type of command that causes no activity in target system, other than a simple response to confirm that it is operational.

any required services an object may depend on (like logging, configuration or similar facilities) with an interface[13]. Then, the interface's requirements are fulfilled by a service externally provided to the object by the framework, when the object is created. [16]

This means that the object no longer directly depends on said service, and it can be replaced by a different service for testing or any other purpose. Furthermore, it bypasses the usual requirement for the object to have to configure the service on creation.

The ASP.NET Core framework allows for creation of services of several types, which differ by their lifetime and functionality. Normally, the documentation speaks about transient, scoped and singleton. [16] Those types of services differ between each other by their lifetime. Transitional services are created each time they are requested by any particular object. Scoped services are created for every HTTP(S) request that the API serves, so they retain information if they're needed multiple times for a single client. Singleton services are created once per lifetime of the whole application, and can thus retain a particular state throughout.

A fourth, lesser known type, is a hosted service. Such services run in their separate thread and can thus carry out continuous background tasks. Although they are slightly more difficult to implement, for instance they have to correctly handle their task being started and stopped, they are indispensable to solve problems like those we face in the design of this system. [17]

Below is a list of custom services in the Data Broker API that I implemented in order to accomplish all functionality requirements assigned to me in this thesis.

- **SignalCache** – Holds the most recent signal values in a thread-safe key-value store. (Singleton)
- **Replay** – Manages the recording of incoming signals into a replay file. (Singleton)
- **DriverProfileManager** – Keeps track of the currently selected car driver. (Singleton)
- **ActivatorCommandHelper** – Stores and resolves the CAN Activator module commands. (Singleton)
- **ActivatorAction** – Used to issue commands to the CAN Activator module. (Scoped)
- **HmiAction** – Used to issue commands to the HMI module. (Scoped)

[13] An interface is a construct in the C# programming language. It defines properties and methods, that any class which implements said interface must have. It can be considered as a template for classes.

- **ValidationScript** – Evaluates actions triggered by AI/ML models via IronPython. (Scoped)
- **MiddlewareContext** – Dependency Injection (DI) wrapper for SQLite database context. (Scoped)
- **SignalPersist** – Periodically saves signal values from SignalCache to the database. (Hosted)
- **DriverProfileSelection** – Used to trigger the driver selection screen after the car's engine is started. (Hosted)
- **DaqSslSocket** – Secured TCP listener for CAN data coming in from the DAQ module. (Hosted)
- *DaqSocket* – Unsecured TCP listener for CAN data coming in from the DAQ module. (Hosted) (Not used in the final design.)

The above list of services does not include standard services provided by the ASP.NET Core framework, such as logging, file provider, HTTP(S) client, and others. Compared to the amount of custom services, the Data Broker / Middleware API has quite a low amount of controllers[14]. Those are as follows:

- **UseCaseController** – Provides the endpoints for registered AI/ML models to connect to.
- **HeartbeatController** – Provides simple heartbeat functionality.
- **UseCaseConfigController** – Provides CRUD functionality for management of known AI/ML models.
- **DriverConfigController** – Provides CRUD functionality for management of known drivers.
- **ReplayConfigController** – Provides control of the Replay service mentioned above.
- *SignalsController* – Provides information about stored signals. Used for debug purposes only. (Not used in the final design.)

In order to describe how all the services and controllers fit together, we must start with the most important part of the Data Broker API, which is the part that stores the parsed signals.

[14]Controllers are the objects in a REST API that actually define endpoints which can be connected to by external applications.

### 3.4.1 Architecture of Data Input and Persistence

The most recent version of a signal is always stored in the SignalCache service. This service internally uses the `ConcurrentDictionary` container, which comes from the standard namespace `System.Collections.Concurrent` of the .NET Core framework. I have chosen this container as it will be accessed by multiple background threads, and this means that it needs to be thread-safe.[15]

The SignalCache service needs to be instantiated as a singleton in the framework in order to ensure that the same collection of signals will be available to every object that accesses it. Signals are stored in the dictionary using their names (strings) as dictionary keys, which leaves some room for further optimization, however, it was currently not found to be a detriment to performance.

Data is stored in the SignalCache by the DaqSslSocket hosted service (and historically DaqSocket). SslDataInlet maintains a TCP listening server, and accepts a continuous, SSL/TLS encrypted stream of NDJSON data from the DAQ utility, as was described in the previous chapters. After the data is decoded from the individual JSON strings, it is stored in the concurrent dictionary.

At the same time, a second hosted service, called SignalPersist is running in the background. Its only task is to save signals from memory cache to the persistent SQLite database every second. Furthermore, it restores signals saved from the database into the memory storage in SignalCache when the Data Broker API starts up for the first time. This ensures that, in the case where the whole API is restarted, at least some signal values are immediately available, until the connection with DAQ is re-established.

### 3.4.2 JSON Data Polymorphism in ASP.NET Core

As the signals coming in from the DAQ utility can either have numeric values (decoded from individual CAN messages using a DBC file) or string values (decoded from the BAP protocol messages using the special utility), the Data Broker API needs to handle JSON data containing slightly different contents. This is known as a variant of data polymorphism.

Polymorphism, by itself, is a very important concept in object-oriented programming. It extends on the concept of inheritance by adding two additional aspects: The objects of a derived class can be treated as objects of the base class, and the derived class can override methods from the base class, if they are declared as overridable. [18]

This practice can be used to implement the logic that we have a base class that represents a given signal (called Signal), and classes that are derived from it for signals with a numeric value (SignalNumeric) and signals with a string

[15] Thread safety means that no unintended side effects or errors will be created if the container is accessed from two or more threads at the same time.

value (SignalString). However, we run into a problem when we want to parse JSON data into both said classes at the same time.

Unfortunately, at the time of the writing of this thesis, the library provided for parsing of JSON data by Microsoft in the .NET Core framework does not support polymorphic deserialization, which means it needs to be handled by creating a custom JSON converter class. A change in the .NET Core framework to add this feature has been proposed. [19]

After implementing said custom JSON converter, the other components of the ASP.NET Core framework integrate flawlessly with polymorphic classes and everything functions correctly. I have used this approach twice: First for having representation for different kinds of signals (base class Signal, inherited by SignalNumeric and SignalString, as was mentioned above), and second for having representation for different command parameters when constructing commands for the CAN Activator (base class ActivatorCommandParameter, with many inheritants), which will be described in later sections.

### 3.4.3 Management of Individual Driver Profiles

As was suggested in the section that described the HMI module, it is beneficial for the training of the AI/ML models to differ between individual car drivers. This is quite obvious, as the habits and preferences greatly vary between different individuals.

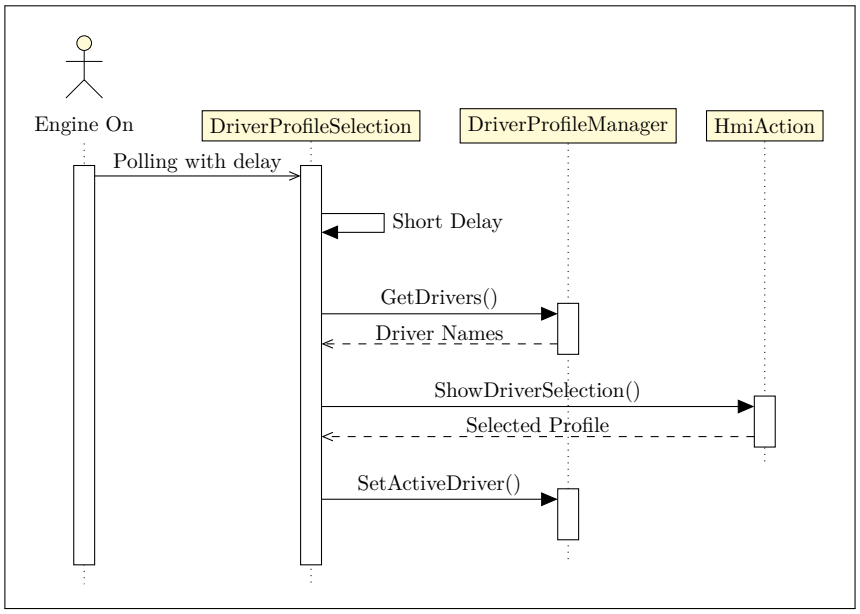


Figure 3.8: Sequence diagram of the Data Broker / Middleware API initial driver profile selection behaviour.

I created a special background service, that is an internal part of the Data Broker / Middleware API, despite behaving in a similar manner to an external

AI/ML model. It periodically checks for the car's engine being started, and it causes the driver selection screen to be shown on the car's infotainment screen shortly after it happens.

The result of this HMI module interaction is then saved to a virtual signal offered by the Data Broker / Middleware API to the individual AI/ML models, named `MIDDLEWARE_Active_Driver_Profile`.

### 3.4.4 Mechanism of Handling AI/ML Model Requests

The Data Broker / Middleware API uses IronPython to enable scripting capability for its AI/ML model controller. The Python script can be customized for every registered AI/ML model, and has two uses. First, it's used to parse data submitted by said model, which enables variability in the submitted data formatting – simply put, the data submitted by an AI/ML model when it triggers an action can be in any form the model (or rather, its creator) desires. Second, it's used to validate said submitted data, in case of the AI/ML model creating nonsensical output (such as setting the air conditioning to 0°C or cruise control to -10 km/h).

This is accomplished by the ValidationScript service, which is instantiated as a singleton, and injected into the UseCaseController, as mentioned previously. When the AI/ML model submits data to execute an action, the controller hands the data over to the ValidationScript service, along with information about which exact model submitted the data.

The ValidationScript service creates a new IronPython script scope[16] and adds the submitted data into the scope as a variable called `CAR_model_data` as it came. Also, it adds all the car signals normally available to the aforementioned model as variables called `CAR_signal_xyz`, where xyz is the original name of the signal.

The script is then executed, and the Data Broker API expects it to create several new variables, which are then read out. The variables are as follows:

- `CAR_result` – represents the general decision of the validation script in whether or not to carry out the action.
- `CAR_result_action_name` – represents the name of the action which will be passed to the CAN Activator module.
- `CAR_result_action_params` – represents the parameters of the action which will be passed to the CAN Activator module, which are expected to be encoded in the JSON format.
- `CAR_result_action_question` – represents the question which will be displayed on the car's infotainment screen via the HMI module.

[16] A scope in IronPython can be explained as an environment for the script to execute in.

The `CAR_result` variable is expected to be a boolean value (yes / no), the rest of the variables are expected to be strings. Only if the script produces all of these variables and the aforementioned boolean result evaluates to true is the action considered to be "confirmed" by its validation script.

Subsequently, the action parameters are validated. This is accomplished using the ActivatorCommandHelper singleton service. How exactly does this service validate the commands for the CAN Activator? At this point we need to speak about one thing regarding Ing. Jaroslav Beran's module. That is, that the module uses a JSON command schema file in order to configure itself. This file can be found in the source code of the CAN Activator in the `config` directory. I copied it to the Data Broker API's source code as well.

When the Data Broker API starts up, the ActivatorCommandHelper service parses this file, and stores in itself the definitions of possible commands and their parameters. Referring to the previous section, the CAN Activator uses JSON polymorphism in its schema quite extensively, and thus another custom converter is required to parse this file.

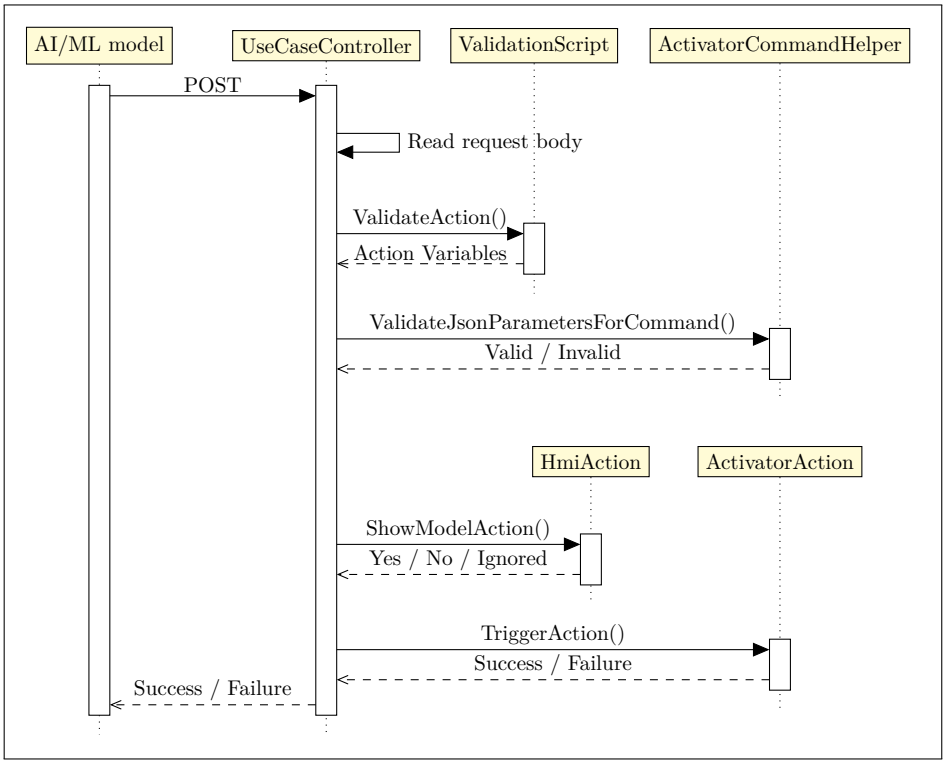


Figure 3.9: Sequence diagram of the Data Broker / Middleware API model data handling behaviour.

After we have validated that the action is indeed correct and all the used mechanisms have presented us with valid data, we can ask the car's driver and / or passengers if they would like to have the action carried out. This is

accomplished by a simple scoped service called HmiAction, which conveniently wraps the communication with the HMI module through the Car-Dependent API using a HTTP(S) request.

If the HMI module is not busy with a previously submitted AI/ML model action, and if the action is indeed allowed by the car's occupants, that is, by tapping on the 'Yes' button presented to them on the infotainment display inside the car, only then, the validated action name and action parameters are handed over to the CAN Activator module. This is fulfilled by the ActivatorAction scoped service.

The result of this chain of events is communicated back to the AI/ML module as a response to its original HTTP(S) POST request to the UseCaseController. This means that action success or any kind of failure (validation script refusal, user refusal, infotainment screen being busy) can be used to further train said AI/ML model, as long as it is capable of waiting some seconds for its HTTP(S) request to complete.

Below is a diagram describing the call hierarchy used to handle an action submitted by an AI/ML model. Please note that the AI/ML models are called 'UseCases' internally in the source code of the Data Broker / Middleware API.

### 3.4.5 Example AI/ML Model Placeholders

As the AI/ML models were not available yet at the time of the writing of this thesis, I developed my own placeholder programs with basic functionality. They act as surrogates for the real AI/ML models in order to test out the functionality of the whole system. I wrote them as Python scripts, once again because of its ease of development, namely due to the Python standard library containing functions for HTTP(S) requests and JSON data parsing.

The first model that I developed is intended for demonstration with the vehicle being stationary. It periodically checks only two signals – the engine being on, and the front passenger seatbelt being fastened. When this situation occurs, an action is triggered that offers to increase the temperature of the front passenger's air conditioning zone.

The second model demonstrates control of a non-touch screen function, namely the adaptive cruise control. It periodically checks the signals indicating whether the engine is on, the GPS[17] coordinates, and movement speed. When the vehicle is reasonably close to an entry to the Prague ring motorway, and driving at a speed that indicates it might indeed be entering said motorway, an action is triggered that offers to set the adaptive cruise control speed to 110 kilometers per hour.

The third model is quite similar, as it demonstrates control of the speed limiter. It once again checks the GPS coordinates, engine status and move-

[17] Global Positioning System

ment speed. When the car seems to be in an average speed check zone near our university's campus, it offers to set the speed limiter to avoid traffic fines.

Despite the popularity of the Python programming language in AI/ML and data science usage, the above models do not contain any such functionality and only rely on imperative programming. Despite this, they create sufficient proof of the whole system's functionality. This shows the system's flexibility and also is an interesting point to ponder. Will AI/ML be truly required in future production use, or is there a possibility that even purely imperative (meaning non-AI/ML) models still positively influence the vehicle's comfort and safety?

### 3.4.6 Example API Endpoint Requests

Below are some HTTP requests that are used to interact with the Data Broker / Middleware API. Configuration endpoints are used, for instance, in order to create an API endpoint for a new AI/ML model or to configure the names of individual driver profiles.

First, here is how a new AI/ML model is registered:

```
POST /api/config/usecase
Request body:
{
    "name": "aircondition_sample_model",
    "signalNames": [
        "X_Kl_15",
        "Y_Seatbelt_Pass"
    ],
    "limitationScript": "
        CAR_result = True;
        CAR_result_action_name = 'air_temperature';
        CAR_result_action_params = '{\"value\": 25, \"zone\": 1}';
        CAR_result_action_question = 'Do you want to set passenger
        air temperature to 25?'
    "
}
```

We can see that a whole short Python script is submitted inside the `limitationScript` variable in the POST request. Model registrations can also be updated by issuing PUT requests, read by issuing GET requests and deleted by using DELETE requests, all on the same `/api/config/usecase` endpoint.

The registered AI/ML model can then access its respective signal values that it is subscribed to on the `/api/usecase/[name]` endpoint. By issuing GET requests, the model receives the values of its signals, and by issuing POST requests, the model can issue its respective action to happen.

```
GET /api/usecase/aircondition_sample_model
Response body:
[
    {
        "name": "X_Kl_15",
        "type": "Numeric",
        "time": 1641316253945,
        "value": 1,
        "unit": ""
    },
    {
        "name": "Y_Seatbelt_Pass",
        "type": "Numeric",
        "time": 1641316253919,
        "value": 0,
        "unit": ""
    }
]
```

The contents of the `limitationScript` variable in the POST request found on the previous page suggest another interesting point to think about. While the AI/ML model is free to send any data it desires with its own POST request to the `/api/usecase/[name]` endpoint, it can also send no data at all, and rely on its validation script *alone* to communicate the desired action to the Data Broker / Middleware API. This is essentially helpful in the case of simple models (like the presented sample ones), as the issued action can simply be hard-coded, and will always be the same.

The Data Broker / Middleware API also has endpoints that allow configuration of driver profiles. A new name can be set for each of the five possible driver profiles by issuing PUT requests on the `/api/config/driver/[id]` endpoint. Please note that the IDs do not belong to abstract entities, they range from 1 to 5, corresponding to the possible buttons on the driver profile selection screen (as seen in the HMI module).

```
PUT /api/config/driver/1
Request body:
"Jan"
```

The list of current driver names can be retrieved with a GET request on the `/api/config/driver/all` endpoint, and also the currently active driver profile can be set and retrieved by issuing POST and GET requests respectivelly on the `/api/config/driver/active` endpoint, as shown on the next page.

```
GET /api/config/driver/all
Response body:
[
    {
        "id": 1,
        "name": "Jan"
    },
    {
        "id": 2,
        "name": "Jiří"
    },
    {
        "id": 3,
        "name": "Lukáš"
    },
    {
        "id": 4,
        "name": "Anne"
    },
    {
        "id": 5,
        "name": "Benji"
    }
]
```

As all the components of the whole system described in this thesis natively support and use the UTF-8 encoding, diacritics and other special characters function flawlessly throughout.

```
GET /api/config/driver/active
Response body:
{
    "id": 1,
    "name": "Jan"
}
```

Another possible way to configure the Data Broker / Middleware API is to use a database editing tool. This is simple, as it utilizes an SQLite database, which is entirely stored in a single file, which means that such database files can be freely copied, externally modified with easily obtainable software, or even exchanged between vehicles.

Chapter 4

# Framework Reliability and Failure Mitigation

Because the system is going to be embedded inside a vehicle and it's not going to have a form that regular users are acquainted with (such as a keyboard, a mouse, or a screen, like a regular computer), manually restarting it in case of failure or strange behaviour is out of the question. For this reason, I carefully studied possible failure points and established how to restore operation if such a failure arises.

| Failure | Resolved By |
|---|---|
| Complete OS freeze or crash | Hardware watchdog timer |
| Data Broker API crash | Init system |
| Data Broker API freeze | Custom heartbeat script |
| Car-Dependent API crash | Init system |
| Car-Dependent API freeze | Custom heartbeat script |
| Activator module crash or freeze | Car-Dependent API scheduler |
| HMI module crash or freeze | Car-Dependent API scheduler |
| DAQ module crash or freeze | Car-Dependent API scheduler |
| BAP utility crash or freeze | DAQ module |

Table 4.1: Investigated Failure Modes of the System Along With Facilities to Resolve Them.

The above table 4.1 mentioned several new terms in the right column that I will now describe in greater detail. The hardware watchdog timer is a feature that some computer motherboards have (including our Advantech industrial PC), and its job is to autonomously trigger a reboot in case the OS becomes dysfunctional. More attention will be dedicated to it in the next chapter.

The init[18] system (short for initialization) is a core component of the Linux operating system. Its task is to start and maintain system services. [20] As the Data Broker API and Car-Dependent API are, in the current implementation, running on the same machine, both need to be started by init when the computer boots up. This also means that in case they crash or exit unexpectedly, the init system will ensure they are restarted.

However, in case the aforementioned Data Broker or Car-Dependent API freeze without exiting or crashing, there is no way for the init system to detect this situation. For this reason, I implemented a heartbeat endpoint in both of their HTTP(S) REST APIs, and added a simple Python script, also running as a system service, that periodically polls these heartbeats with a certain, sufficently high timeout. If the polling fails, the script notifies the init system that the respective API needs to be restarted.

As was mentioned in previous chapters, the HMI, DAQ and Activator modules all are launched as subprocesses of the Car-Dependent API, and as such are monitored for crashes or freezes by it. The BAP utility is a subprocess of the DAQ module, and therefore is under its control.

## 4.1 Testing of Web Framework-Based Components

Looking back behind us we find an amusing situation. Despite this whole thesis being about embedded or semi-embedded software, its two most important parts have been made using frameworks normally intended for creating websites. Evaluating the requirements laid out at the assignment of this thesis, this is, fortunately, far from preposterous in my case.

Because the reliability and safety requirements are, in our case, fairly high, additional care should be taken regarding testing and analysis specific to web frameworks and REST APIs. As rigorous testing is not, unfortunately, a part of my specialization, I focused my attention to static testing techniques, such as informal review [21] and reporting to and consulting with my supervisor about modes of operation and reasoning behind my API design. Requirements regarding outside-facing APIs were also discussed with the car manufacturer's employees.

Static analysis and evaluation against best practices of API development, both regarding the C# with the ASP.NET Core framework, as well as the Python language with the Flask microframework, was carried out to the best of my capabilities, however, as this thesis and the system is presently developed solely by me, there is definite room for consultation and improvement.

Further, more formal, testing is intended to be carried out later in the lifetime of this project, when there will be more people working on it. This will enable effective use of dynamic analysis and testing techniques, such as unit testing or acceptance testing. This has been taken into account in the

[18]The name is not supposed to be capitalized.

development of all major parts of our system – correct use of object-oriented programming prepares us for easy integration with dynamic tests.

Reliability will be further investigated in the following chapter, as the OS environment as well as the computer configuration are inseparable parts of reliability analysis.

## 4.2 Preserving Partial Functionality in Case of Errors

As was mentioned in the previous chapters, the Data Broker API persistently saves signal values every second to its SQLite database. This is so that at least some data is available if there is an error on the CAN interface, or during initial startup when the communication with the DAQ module may not be established yet.

Addtionally, it is easy to imagine that some basic AI/ML models may live on the embedded computer inside the car itself, so that some operation of the system is preserved if connection to the internet is lost. Otherwise, due to the HTTP(S) REST methods of communication between higher-level components of the system (including the external AI/ML models), temporary loss of connectivity to the outside world doesn't pose any major problems.

However, if the Data Broker / Middleware API is to be decoupled from the car, care needs to be taken so that the streaming NDJSON over SSL connection from the DAQ module is restored. This can be accomplished by implementing functionality to resend a DAQ configuration request to the Car-Dependent API in case of no data being received by the aforementioned DaqSslSocket service in a certain time period.

## 4.3 Custom Heartbeat Script

As was mentioned in table 4.1 and surrounding paragraph, a new facility is required to guard the Data Broker API and the Car-Dependent API in case of any possible freezes. While a crash would be resolved by the operating system's init system,[19] a freeze needs to be proactively checked for.

Both the APIs have a heartbeat endpoint, which simply returns a HTTP OK status when a GET request is issued. A simple Python script, which itself is also going to be run as a system service, periodically polls these endpoints. In case a request times out (signalizing a freeze has occurred), the script issues a command to the init system to restart the offending API.

[19] Because the Data Broker API and Car-Dependent API are going to be run as system services.

Chapter 5

# Software and Hardware Configuration for Optimal Operation

The system was developed, for the most part, on the embedded computer, which is running the Ubuntu 20.04.3 LTS distribution of the Linux operating system. This was done to ease development efforts, as the computer was interacted with in many different ways, including physical access to the desktop enviroment via a temporarily connected mouse, keyboard and screen, but also using remote access, both via VNC and SSH[20].

For future use, it is easy to imagine a more streamlined operating system being used instead. Options include the Ubuntu Server distribution, or a completely custom build of embedded Linux. Ways to accomplish building a custom version of embedded Linux include the Buildroot and Yocto frameworks. As such a task was not assigned as a part of this thesis, I haven't dedicated much time to investigating it further. Nevertheless, it is clear from an engineering perspective that this point will need to be investigated further in the lifetime of this project.

## 5.1 Operating System Environment

The operating system running on the embedded computer must support running all the parts of the system / framework described in this thesis. This includes the following runtimes, frameworks and libraries:

- ASP.NET Core Runtime version 5.0 or higher.
- Python version 3.7 or higher.

[20] VNC is a protocol used for remote desktop access, while SSH is a protocol used for remote command line access.

- Flask microframework version 2.0 or higher, including the Flask-RESTful extension (installed through Pip package manager).
- APScheduler version 3.0 or higher (installed through Pip package manager).
- Qt library version 5.15 or higher, including the Serial Bus and Network components.
- libssl1.0.0 library (exact version required for the BAP utility).

The last mentioned point in the above table is quite curious, and displays the downsides of closed-source software. The libssl1.0.0 library is no longer a part of the repositories on the Ubuntu distribution in the version 20.04 LTS and above. This means that the package either needs to be installed from a custom repository, or manually imported. If there was a possibility to easily recompile the BAP utility from source, a newer version of the libssl library could be used.

## 5.2 Leveraging the Embedded PC Watchdog Timer

During the testing and troubleshooting of the system running physically inside the vehicle, a new issue was uncovered. Despite the arguably quite robust DC/DC converter used to power the computer, changes in the supply voltage caused by the operation of the vehicle (ex. engine cranking) can have impact on the function of the computer, even leading to lock-ups in rare cases. Industrial computers provide hardware functionality that is perfectly suited to mitigate this problem – the watchdog timer.

The watchdog timer functions as follows: When activated, it runs a countdown for a set amount of time (ranging from seconds to minutes). When this countdown is over, the watchdog timer generates a reset signal. This countdown is periodically reset back to start by the software of the industrial computer. Thus, if the software (or operating system) crashes or freezes, it stops resetting the timer, the timer runs out and the whole computer restarts, allowing for autonomous recovery. Coupled with fast boot-up times enabled by optimized operating system and solid-state storage, this makes for a significant boost in the computer's robustness.

Sadly, the watchdog timer contained in modern Advantech computers seems to be non-standard and unsupported by the usual facilities of the Linux kernel (namely `watchdog(8)` and its device file `/dev/watchdog`). Although the manufacturer provides raw assembly code in its user manual that shows how to control the timer through the I/O ports of the x86 platform, very little else is explained on how it actually works, or even through which chip the functionality is implemented. [14]

Additional research on this topic is further complicated by the fact that the Advantech ARK-3520P computer actually contains several components which are capable of providing the watchdog functionality. Those are the computer's two different (!) Super I/O chips, the Intel Management Engine (IME) found on the computer's chipset, and/or possibly a different component, called Embedded Controller[21] by the manufacturer.

The first three of these components all have some degree of support for the watchdog functionality in the Linux kernel. Those are:

- The first Super I/O chip (IT8768E), partially[22] supported in the kernel by the `it87_wdt` module.
- The second Super I/O chip (NCT6106D), fully supported in the kernel by the `w83627hf_wdt` module.
- The Intel IME/AMT OS Health Watchdog, fully supported in the kernel by the `mei_wdt` module. [22]

Unfortunately, none of those modules seem to be functional on our particular computer. While the first two modules seem to enable the `watchdog(8)` daemon,[23] manually causing a deliberate reboot by issuing the command:[24]

```
# cat >> /dev/watchdog
```

sadly did not work. Loading the `mei_wdt` module did not even seem to enable the daemon. These findings suggest that none of the first three watchdog timer variants are utilized in our particular computer. Furthermore, neither of the three kernel modules in the list above use I/O port adresses that correspond with what was described in the computer's manual.

Further researching the user manual for the Advantech ARK-3520P computer, I came across mentions of Advantech's management software called iManager together with a library/API called SUSI 4.0. Their website does indeed offer a download link for the SUSI 4.0 product together with example programs, however, the included library is only for the Windows operating system. Moreover, the datasheet of the computer states that Linux support is "By Project Support". [5]

All that is certain is that software for this computer targeted at the Linux operating system is nowhere to be found on the manufacturer's website. I

[21] This component will be mentioned several times in this section. It seems to be referred to as 'Embedded Controller', 'EC', and possibly 'AHC1EC0' by the manufacturer, although the author has no concrete confirmation if these terms are equivalent to eachother or not.

[22] The driver in its current form doesn't recognize the chip ID, however, support would be trivial to implement according to existing driver functionality and available datasheets.

[23] A daemon is a term for background processes in Unix-like environments.

[24] This command precludes resetting the watchdog timer countdown on systems that have the `watchdog(8)` daemon running and functional.

would like to reinforce the point that I would prefer to have an actual watchdog timer driver that conforms to the methods that Linux uses to manage the timer's functionality – e.g. a driver/module that behaves like the three modules mentioned in the paragraphs above. That is, enable the use of `watchdog(8)` daemon and the `/dev/watchdog` device file. The reason for my preference is cleanliness of implementation, maintainability and an overall wish to keep things standard.

Since the raw assembly sample code is mentioned under the name "EC Watchdog" in the computer's manual, and because the computer's BIOS setup has the options to configure the watchdog timer under a page called "Embedded Controller Configuration", I researched keywords such as "Advantech", "EC", "watchdog", "WDT" and "Linux" on the internet. This led me to a website containing the patch tracking system of the Linux HWMON (Hardware Monitoring) subproject.

On this website, there is a proposed change to add the support for a chip or a device called **Advantech AHC1EC0**. This seems to be what we are looking for! Further investigation of the source code additions contained in this proposal reveals a watchdog timer driver along with further hardware monitoring functionalities. [23]

Unfortunately, there seem to be disagreements on the particularities of the methods that Advantech's hardware uses to interface with the Linux kernel (namely the usage of ACPI, versus device-tree), and the changes have been rejected for now. Further revisions that will rectify these disagreements appear to be in development by Advantech or their employees, however, at the time of the writing of this thesis they were not finished.

Thus, we arrive at a choice between two options: Program our own simple utility using the information provided in the Advantech ARK-3520P user manual, or attempt to extract the source code for the AHC1EC0 watchdog timer kernel module (`ahc1ec0-wdt.c`) out of the proposed patch and attempt to build it out-of-tree for our own kernel.

For the sake of simplicity, I elected to explore the simpler option first, as replicating one short page of assembly code from the computer's manual seemed an easier way to accomplish the task, notwithstanding the reasoning in the above paragraphs. I ported the source code found in the appendix A of the user manual [14] to the C programming language and compiled it on my Linux system. I executed the resulting program and... nothing happened.

At this point we must remember that the option to enable the watchdog timer itself is found in the BIOS setup! However, here I uncovered two new fun complications. An option to configure the watchdog timer can be found in the setup page called "AMT Configuration" (for the Intel Management Engine (IME) technology), **but also** several setup pages later, on a page called "Embedded Controller Configuration".

I have previously experimented with the watchdog timer contained in the Intel Management Engine part of the computer's chipset and established it to

not be functional, in spite of it being enabled in the BIOS. So, we turn our attention to the "Embedded Controller Configuration" page. This seems to correspond with the name given to the watchdog in the computer's manual ("EC Watchdog").

After enabling this option, two more configuration options show up, **which are not documented in the user manual**! Those are to set the watchdog timer's duration and a selection between minutes and seconds for said duration. From this I inferred that the watchdog timer gets started by the BIOS upon power on and starts ticking immediately. A quick reboot and a short wait for the computer's automatic restart caused by the watchdog timeout confirmed my suspicion.

Hastily running my utility after the operating system booted up did indeed save the computer from restarting. However, further experimentation showed that the timeout duration stayed the same as was set in the BIOS setup. This is in spite of the reference source code supplied by the manufacturer supposedly having the capability of setting its own timeout. The only function of said code that was shown to reliably work was the function to (re)start the timer and prevent a reboot caused by a timeout. Amusingly, this is accomplished by the three last assembly code instructions alone; all the other instructions from the manufacturer's manual were not used in the final version of my utility.

At long last, the computer's watchdog timer is working. I fashioned my utility to restart the watchdog timer at configurable intervals and to be run as a systemd[25] service. I configured it to start at the `sysinit.target` part of the operating system start-up procedure (thus, quite early). With the watchdog timer timeout set in the BIOS to 60 seconds, and with my utility configured for timer restarts at 55 second intervals, the operating system has plenty of time to boot up, and I have finally achieved a more robust system.

## 5.3 Ensuring Reliable Startup of the System

We have previously established in this thesis, that the two (independent) components that need to be started up by the operating system are the Car-Dependent API and the Data Broker / Middleware. Same as the EC watchdog control utility described in the above section, these two components will need to be launched as services, by the computer's init system.

This adds an additional reliability benefit in that the init system monitors the health of those two main processes, and restarts them if necessary. As we can consider the Data Broker / Middleware and the Car-Dependent API components of the computer's high-level functionality, I configured them to start at the end of the computer's start-up procedure, namely at the `multi-user.target` part.

[25] systemd is a popular init system (service manager) for the Linux operating system. Its name is not supposed to be capitalized.

Another particularity of the Linux operating system and its SocketCAN component is that the CAN interfaces are, by default, not brought up and configured when the computer starts, and need to be configured manually. I once again leveraged the systemd init system to configure the two CAN interfaces to start autonomously and thus enable the correct function of the whole system.

I created two configuration files (one for each respective CAN interface) in the `/etc/systemd/network/` directory. The files are called `80-can0.network` and `81-can1.network`. These files contain information on how to configure the two CAN interfaces and how long to wait before restarting the interfaces in the case of an error. [24] Now, as long as the `systemd-networkd` service is enabled, the aforementioned directory will be processed and the configuration from the files contained in it will be applied, in the order of the numbers contained at the start of their filenames (80 and 81 in our case).

In case the whole system is to be recreated using a more embedded Linux distribution,[26] which may not use the systemd init system, these configuration files will need to be translated to the new init system of choice (such as SysVinit or Busybox init). As no systemd-specific features are leveraged by my software, this would be trivial.

[26]Such as those created by aforementioned Busybox or Yocto frameworks.

Chapter 6

# Conclusion

Finally, the whole system is complete. As was assigned for me to do, the framework acts as a middleware between the hardware layer and the AI/ML model layer. The models communicate using the HTTP(S) protocol, adhering to the REST API methodology.

The system is reasonably tolerant to faults, utilizing both hardware and software mitigations for errors, and caching in cases of communication loss. It has also been designed to be modular, with the possibility of moving some parts of it from the vehicle to the cloud. There is also a possibility of doing the opposite, and moving simpler models to the vehicle itself for enhanced reliablity.

Functionality has been evaluated and proven using non-AI/ML scripts in place of a model, establishing the functionality of outside-facing API but also showing that simple predictions can be made without the use of artificial intelligence, with imperative programming (if-this-then-that) being sufficient in simpler use-cases.

Implementing and connecting together such a diverse system with several different components was an immense learning experience, and in my opinion, makes a good example of my specialization (Computer Engineering), especially for other people with an IT or EE background.

# Glossary

**BAP** Bedien und Anzeigeprotokoll, high-level protocol for automotive networks such as CAN, LIN or FlexRay.

**Boost** Boost, is a free, portable and peer-reviewed library of functions for the C++ programming language with an extremely wide range of uses.

**CAN** Controller Area Network, robust communication network most commonly used in automotive and industrial applications.

**CAN FD** Controller Area Network Flexible Data-rate, an extension to the CAN protocol allowing faster data rates to be used and larger message sizes to be transmitted.

**CMake** CMake, is a cross-platform free and open-source software for the automation of compilation, testing, packaging and installation of computer software by compiler-independent methods. By itself it is not a build system, rather it generates files for a pre-existing build system of the user's choice, such as Make, Qt Creator, Ninja, Xcode and others. [25]

**CP1252** Code Page 1252, is a character encoding used in some older versions of Microsoft Windows to encode English and some other western european languages, such as German, Spanish or French.

**IronPython** IronPython, is an open-source implementation of the Python programming language built within the .NET framework. It allows accessing .NET functions and API from Python scripts, but more importantly, also the execution of Python scripts within a .NET application.

**JSON-RPC** JavaScript Object Notation - Remote Procedure Call, is a lightweight protocol for the exchange of remote procedure calls, that is, invoking methods in a remote program (peer) and the exchange of method parameters and results.

**NDJSON** Newline-delimited JavaScript Object Notation, is a format for transfering separate objects or messages via a JSON stream. It simply consists of individual JSON objects separated by newline characters (ASCII code 0x0A), or in some cases by record separator characters (ASCII code 0x1E).

**Pip** Pip Installs Packages, is a package management system for installing additional packages for the Python programming language. Most Python installations now include Pip by default.

**RS-232** Recommended Standard 232, is a standard that defines serial transmission of data between two pieces of equipment. Briefly put, it is the serial port (COM) we know and love from our computers, however, the standard was originally introduced as far back as the 1960s. [15]

**SocketCAN** SocketCAN, is a low level interface for communication on the CAN bus on the Linux platform. It enables CAN interfaces to be configured and interacted with in a similar matter to Ethernet interfaces, via standard socket APIs.

# Acronyms

**ACPI** Advanced Configuration and Power Interface

**AI/ML** Artificial Intelligence / Machine Learning

**AMT** Active Management Technology

**API** Application Programming Interface

**ASCII** American Standard Code for Information Interchange

**BIOS** Basic Input Output System

**CRUD** Create, Read, Update, Delete

**CTS** Clear To Send

**CTU in Prague** Czech Technical University in Prague

**DAQ** Data Acquisition

**DBC** DataBase CAN

**DI** Dependency Injection

**DSR** Data Set Ready

**DTR** Data Terminal Ready

**FEE** Faculty of Electrical Engineering

**GPIO** General Purpose Input/Output

**GPO** General Purpose Output

**GUI** Graphical User Interface

**HDMI** High Definition Media Interface

**HMI** Human-Machine Interface

**HTTP** Hypertext Transfer Protocol

**HTTP(S)** Hypertext Transfer Protocol (Secure)

**ID** Identifier / Identification Number

**IME** Intel Management Engine

**IP** Internet Protocol

**JSON** JavaScript Object Notation

**LTS** Long-Term Support

**LVDS** Low-Voltage Differential Signaling

**ORM** Object-Relational Mapping

**OS** Operating System

**PDF** Portable Document Format

**PS** PostScript

**REST** Representational State Transfer

**RTS** Ready To Send

**SCTP** Stream Control Transmission Protocol

**SSH** Secure Shell

**SSL** Secure Socket Layer

**TCP** Transmission Control Protocol

**TLS** Transport Layer Security

**UDP** User Datagram Protocol

**UI** User Interface

**UTF** Unicode Transformation Format

**VNC** Virtual Network Computing

# Bibliography


[1] N. Nugent, *The Knight Rider Companion*. El Segundo, CA: Will Garris Publishing, 2008.

[2] J. Huth, *Knight Rider Legacy: The Unofficial Guide to the Knight Rider*. Bloomington, IN: iUniverse, Writers Club Press, 2004.

[3] J. Beran, "*Firmware for Control Module of an Intelligent Vehicle*," Master's Thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, 2020.

[4] Department of Measurement, CTU FEE in Prague, *Dokumentace Modulu CAN FD Gateway*. Czech Technical University in Prague, Faculty of Electrical Engineering, 2018.

[5] Advantech Co., Ltd., *Fanless Box PC ARK-3520P Datasheet*, December 2020. https://advdownload.advantech.com/productfile/PIS/ARK-3520P/file/ARK-3520P_DS(121420)20201214160852.pdf.

[6] G. Fu, Y. Zhang, and G. Yu, "A Fair Comparison of Message Queueing Systems," *IEEE Access*, vol. 9, pp. 421–432, 2021. https://ieeexplore.ieee.org/document/9303425.

[7] "*TechEmpower Framework Benchmarks*." https://www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=composite, August 2021. Accessed: 2021-11-30.

[8] JSON-RPC group, *JSON-RPC 1.0 Specification*, 2005. https://www.jsonrpc.org/specification_v1.

[9] M. Luparu, "CMake support in Visual Studio." https://devblogs.microsoft.com/cppblog/cmake-support-in-visual-studio/, October 2016. Accessed: 2021-12-16.

[10] K. Bridge, K. Sharkey, J. Kirsch, and M. Satran, "Unicode." https://docs.microsoft.com/en-us/windows/win32/intl/unicode, January 2021. Accessed: 2021-12-16.

[11] P. Hoffman and F. Yergeau, "UTF-16, an encoding of ISO 10646," RFC 2781, Internet Requests for Comments, February 2000. https://datatracker.ietf.org/doc/html/rfc2781.

[12] F. Yergeau, "UTF-8, a transformation format of ISO 10646," RFC 3629, Internet Requests for Comments, November 2003. https://datatracker.ietf.org/doc/html/rfc3629.

[13] P. Radzivilovsky, Y. Galka, and S. Novgorodov, "*UTF-8 Everywhere*." http://utf8everywhere.org, 2012. Accessed: 2021-12-16.

[14] Advantech Co., Ltd., *ARK-3520P User Manual*, April 2017. https://advdownload.advantech.com/productfile/Downloadfile3/1-1DJVBD8/ARK-3520P_User_Manual_Ed.2-final.pdf.

[15] "EIA/TIA-232-F:1997 (R2012)," Standard, Electronic Industries Association, 2012.

[16] D. Pine, G. Warren, G. Chan, and R. Anderson, "Dependency injection in .NET." https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection, December 2021. Accessed: 2021-12-21.

[17] R. Anderson, R. Larkin, J. Lomholdt, and T. Dykstra, "Background tasks with hosted services in ASP.NET Core." https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-6.0, December 2021. Accessed: 2021-12-21.

[18] B. Wagner and P. Kulikov, "Polymorphism." https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/polymorphism, June 2021. Accessed: 2021-12-21.

[19] E. Tsarpalis, "JsonSerializer polymorphic serialization and deserialization support." https://github.com/dotnet/runtime/issues/30083#issuecomment-861524767, June 2021. Accessed: 2021-12-21.

[20] B. Ward, *How Linux Works: What Every Superuser Should Know*. San Francisco, CA: No Starch Press, 3 ed., 2021.

[21] Y. Boronenko, "*Software Testing in Agile Development Methodologies*," Bachelor's Thesis, Czech Technical University in Prague, Faculty of Information Technology, 2019.

[22] "Intel(R) Active Management Technology (Intel AMT) – The Linux Kernel documentation." https://www.kernel.org/doc/html/latest/driver-api/mei/iamt.html, 2019. Accessed: 2021-12-16.

[23] "Add Advantech AHC1EC0 embedded controller entry – Linux HWMON Patchwork." https://patchwork.kernel.org/project/linux-hwmon/patch/20210506081619.2443-1-campion.kang@advantech.com.tw/, May 2021. Accessed: 2021-12-15.

[24] F. Voorburg, "Automatically bring up a SocketCAN interface on boot." https://www.pragmaticlinux.com/2021/07/automatically-bring-up-a-socketcan-interface-on-boot/, July 2021. Accessed: 2021-12-29.

[25] "About CMake." https://cmake.org/overview, 2019. Accessed: 2021-12-16.

Appendix A

# Contents of the included DVD

/
- `readme.txt` ... Brief description of DVD contents in plain text format
- `readme.pdf` ..... Brief description of DVD contents in PDF format
- `sources`
  - `daq` ..... Source code of the DAQ module
  - `hmi` ..... Source code of the HMI module
  - `activator` ..... Reference to the Activator module
  - `middleware` ..... Source code of the Data Broker / Middleware
  - `car_rest_api` ..... Source code of the Car-Dependent API
  - `watchdog` ..... Source code of the EC Watchdog control utility
  - `dbc` ..... Source code of the DBC library
  - `api_heartbeat` ..... Source code of the API heartbeat script
  - `sample_models` ..... Source code of sample Python AI/ML models
  - `thesis` ..... LaTeX source of this thesis
- `config`
  - `systemd` ..... systemd configuration and service files
- `thesis.pdf` ..... Thesis text in PDF format
- `thesis.ps` ..... Thesis text in PS format