Delft University of Technology
Master of Science Thesis in Embedded Systems

# Next Generation Innovation Vehicle HMI System

**Arend-Jan van Hilten**


TU Delft Delft University of Technology

# Next Generation Innovation Vehicle HMI System

Master of Science Thesis in Embedded Systems

Embedded and Networked Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
The Netherlands

Arend-Jan van Hilten

2023-01-26

**Author**
Arend-Jan van Hilten
**Title**
Next Generation Innovation Vehicle HMI System **MSc**
**Presentation Date**

2023-02-09

**Graduation Committee**

| | |
|---|---|
| Prof. Dr. Koen Langendoen (chairman) | Delft University of Technology |
| Dipl. Ing. Volker Vogel | ZF Friedrichshafen AG |
| Prof. Dr. Ir. Martijn Wisse | Delft University of Technology |
| Martin Klomp MSc | Delft University of Technology |

**Abstract**

Modern vehicles have multiple different buses to communicate between components, like CAN (FD) and FlexRay. ZF builds "innovation vehicles" with new components to showcase and test them. These components are connected to the automotive buses. ZF uses a web-based Human Machine Interface (HMI) to control and view the state of these parts. This HMI is needed because some systems are not visible or controllable in regular operation. A gateway is required to connect the HMI and the buses.

ZF currently uses a CAN to WebSocket gateway that does not support other buses. There is no readily available hardware with the required buses and interfaces. A WebSocket interface is required, as the HMI is running in a browser, limiting the possible protocols.

Therefore, the challenge for this thesis is how to (re)design this HMI system for future innovation vehicles with buses besides CAN. Each vehicle can have a different number and types of buses, so the system must be able to cope with this. The HMI system is not the only part in the network, so it also must be efficient to not interfere with other systems, like network cameras. This redesign was done by analysing the types of buses, the hardware that could be used, and the software components needed for this flexible system. The software components were mapped onto the available hardware to make the architecture as flexible and efficient as possible.

Two mappings are proposed, one using a Software Gateway, a custom application with support for different hardware interface drivers, capable of running on Windows, Linux and in containers. The other mapping uses WebSockify, where WebSocket messages are converted to TCP/UDP messages. These two mappings were combined into a single architecture to combine the features and possibilities of both systems.

Then a Minimum Viable Product (MVP) was made to test the envisioned architecture, showing excellent results compared to the current solution while adding more flexibility and other features.

Using this new HMI system, the HMI developers can interface with more types of networks, build HMIs that connect to multiple different vehicles and make distributed HMI systems.

# Preface

I have always been interested in cars, primarily Volvo classic cars, which resulted in my own 1967 Volvo Amazon. I also support my father's journey of converting a 1982 Volvo 245 to full-electric. Thanks to him, I was introduced to Volker Vogel for a thesis project. I was looking for a practical project, and ZF had this opportunity in Friedrichshafen to work on the Innovation trucks and do a thesis afterwards. This resulted in me living for ten months in Germany, where I gained a lot of knowledge about vehicles and automotive buses and met many international friends.

I will take this knowledge with me for future projects, and I will also use this to modernise my classic car!

I want to thank Volker Vogel for the opportunity and his outstanding mentorship and support these past few months. I also want to thank all the ZF colleagues I had the joy of working with during my time in Friedrichshafen and Hannover. Lastly, I want to thank Koen Langendoen for his trust and for coaching me on the journey of creating this thesis out of the project.

Arend-Jan van Hilten

Delft, The Netherlands
26th January 2023

# Contents

# Chapter 1

# Introduction

ZF Friedrichshafen AG (ZF) is an international supplier of parts and systems for personal and commercial vehicles and other industries like maritime and wind. This includes transmissions, airbags, clutches, seat belts, emergency braking systems, and many others.

When ZF has a new or improved part that they want to test or showcase to customers, they sometimes build an innovation vehicle. This is a normal car or truck that is retrofitted with the new part and other tools to monitor and test the new elements. A few examples of recent innovation vehicles include a Volkswagen Touran with rear-axle steering, a truck-trailer combination fitted with all kinds of efficiency-optimizing parts, like a regenerative braking trailer with aerodynamic flaps, and a truck-trailer system focused on safety, with systems like emergency braking, lane keeping and seat belts with haptic feedback.

Some of the new parts are easily visible, like the aerodynamic flaps, and can be easily shown to anyone interested. The systems like emergency braking and lane keeping can also be shown quite easily but require some input to enable the (experimental) features for safety. Other systems like the rear axle steering and regenerative braking are not that visible or controlled easily and need some way for the driver and passengers to be enabled or controlled to be visible or experienced.

All these systems benefit from an Human Machine Interface (HMI) to control and show the workings of the new parts. All of the previously mentioned vehicles got an HMI system to control the parts, even in ways they should not be used in real vehicles. The regenerative braking trailer, for example, is controllable with a slider to brake or push the truck to let the passengers feel the power of the system. The motorised seat belts are usually not controllable by the driver and should only give a haptic warning when there is an unsafe driving situation. However, in the safety truck, it is possible, using the HMI, to send a command to the retractors to give a haptic warning or pull the webbing at a high force.

Most of the new innovative parts use Controller Area Network (CAN) to connect to the rest of the vehicle. All modern vehicles have multiple of these standardised network buses, so connecting the new part is no problem. Connecting an HMI to this bus is more of a challenge because the HMIs are built as a web app and can be run in any browser on any hardware, like tablets, phones and computers. These devices do not have a CAN interface, so a hardware gateway with CAN is required. Running the HMI in a browser creates

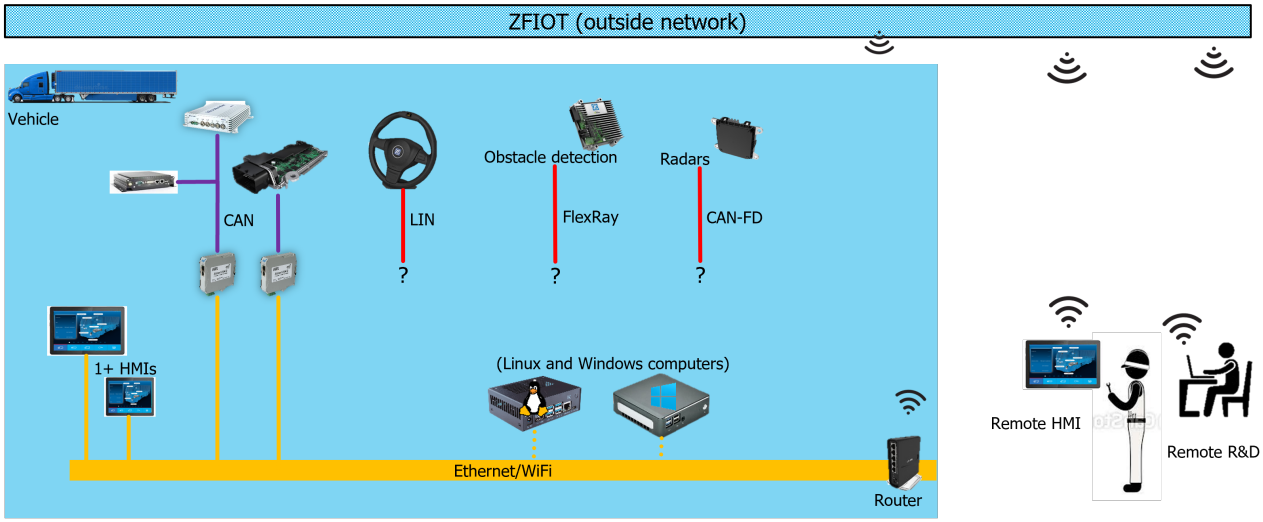



Figure 1.1: **Network layout for the innovation vehicles with a variety of different buses and connected hardware**

another hurdle, as websites are not allowed to connect to any internet socket, but only HTTP requests and WebSocket connections are allowed. Therefore the hardware gateway needs a WebSocket server. The possible hardware with CAN (FD) interfaces that could be used often does not have this application programming interfaces (APIs), so an intermediate system needs to be built to support the assortment of hardware and to make it efficient. The system needs to be efficient, as the Ethernet network is also used for other purposes, like video streams from network cameras, and it should be possible to have a significant amount of HMIs connected while keeping the resource usage low.

The current HMI system for the innovation vehicles uses Android tablets for the driver and the passengers to interact with. These tablets run a web app in a full-screen browser. This makes it easier for the developers to build the HMI and makes it possible to be used on any device that can run a browser, even in another network, and it can be tested outside of the vehicle. Tablets are used instead of laptops or industrial HMIs to give it a more innovative and modern look.

The current system layout is shown in Figure 1.1. The tablets are connected through Wi-Fi to a router. The router is connected to the outside network through Wi-Fi as well to allow remote development and HMIs running outside the vehicle. Sometimes a Windows or Linux computer is mounted inside the vehicle for the innovation parts, like an Nvidia Jetson Nano for object detection. Gateways are added to connect to the vehicle buses. The current system uses the *ESD* EtherCAN/2 that has a WebSocket interface. The web app on the tablets connects to this interface. On the CAN, all kinds of electronic control units (ECUs) are connected and added.

This current system has worked for over ten years since the first HMI with this system was made with it in 2012. In the meantime, other buses, like Controller Area Network with Flexible Data-rate (CAN FD), Local Interconnect Network (LIN) and FlexRay, have emerged to improve speed, increase message data

size, improve timing constraints and reduce cost. These message buses can all be converted to CAN by using an extra gateway, but this requires additional parts, extra configuration and most importantly, more space, which is often constrained, especially in passenger cars. A new gateway with all those buses and a WebSocket interface would be perfect, but it does not exist yet.

## 1.1 Problem statement

No system was found that can interface with all the different buses while providing a WebSocket interface. Therefore a new gateway system for the next-generation innovation vehicles must be designed to interface with all the different new network buses. The system must also support specific software features (see below) to be better usable for an HMI. The challenge for this thesis is thus

> How to (re)design a Human Machine Interface (HMI) system for future innovation vehicles.

The requirements for this HMI system, as defined by ZF and by inspecting current HMIs, are as follows:

1. Vehicle Connectivity:
   (a) CAN bus
      i. Normal & CAN-FD speeds
      ii. 11-bit & 29-bit message IDs
      iii. Configurable speeds
   (b) LIN
   (c) Preferred: FlexRay & Automotive Ethernet
2. HMI Connectivity:
   (a) WebSocket server:
      i. Bi-directional and multiple HMI connections
      ii. Message decoding module in JavaScript (JS)
      iii. Filtering on message/bus, whitelisting
      iv. Efficient messaging, bundling of messages
      v. downsampling of messages
      vi. <75 millisecond (ms) (13 Hz) roundtrip time
   (b) HTTP-server:
      i. Serve HMI and debug pages
      ii. >500MB size possible
3. Configurability:
   (a) Modular system
   (b) Centralised
   (c) Network settings
   (d) Website for configuration
   (e) HTTP-server settings
4. Hardware:
   (a) Production type hardware
   (b) boot-time: <30s
   (c) Must have a housing
   (d) Size: 20x30x10cm to fit easily in vehicles
5. Other:
   (a) Website with state of different modules
   (b) Supply voltage 10-32V
   (c) Per-project budget
      i. Small HMI project: <€1k
      ii. Big HMI project: <€5k
   (d) No software licensing per developer or device

Requirement 2(a)vi is there to have a fast enough reactive system without pushing the network and devices to the limit. 13 Hz is the standard human visual refresh rate, so anything more rapid than that is sufficient[27]. Getting more data only puts more load on the tablets, network and intermediate components. The new HMI should be easily configurable, preferably from a centralised point, like the HMI code. A centralised configuration should make it possible to change hardware and lets HMI developers test outside the vehicle.

The challenges of this project were the interfacing hardware, how to communicate with them, which possible hardware exists for the Software Gateway, could virtualisation help for this project, how to design an (software) architecture for the complete system and implementing that for the Minimum Viable Product (MVP).

The types of buses are discussed in Section 2.1, along with virtualisation in Section 2.3 and other messaging systems in Section 2.4. The current system that has worked for over ten years is looked into in Chapter 3. In Chapter 4, the global architecture is discussed. The interfacing hardware is researched in Appendix A to look into the possible hardware, which hardware is already in use and which hardware is interesting to add. In Section 5.1, the architecture of the MVP is shown. The gateway hardware and types are discussed in Section 5.4, and the specific architectures are shown for each hardware interface in Appendix A.

# Chapter 2

# Background information and existing systems

As discussed in Chapter 1, multiple types of automotive buses are in use in modern vehicles. These are explained in Section 2.1, and a conclusion and implications for the HMI system are drawn in Subsection 2.1.6. Besides the vehicle communicating with the system, the HMI must also communicate with the rest of the system. The types of communication that can be used for the HMI-gateway communication are listed in Section 2.2. Then Docker and its advantages are explained in Section 2.3 as it helps deployment to gateway hardware and allows running code on the routers in use. In Section 2.4, some other message based systems are discussed and their use cases. To conclude, Subsection 2.4.4 explains what can be taken from those systems and why they cannot be used for the HMI system.

## 2.1 Automotive Networks

Where oldtimers only have a simple radio, some lights, and a distributor as electronics, modern cars have many more systems on board. From motor control to safety systems and entertainment systems to even vehicle-to-vehicle communication in the future. All these systems need to communicate with each other. In classic cars, there are only some analogue signals and possibly some digital signals. These 'protocols' are not sufficient for more modern cars. Therefore multiple different communication buses were invented. All modern cars have multiple of these buses, and often multiple types are used for different purposes. In the following sections, a few buses will be explained, and a lookout on future network protocols.

These communication buses operate on the data-link layer and physical layer of the ISO model[3]. Often no other extra layers are in-between the application and the data-link layer, but protocols like SAE J1939 can be used.

### 2.1.1 CAN

The CAN bus is a multi-master network that uses a single twisted pair of wires, consisting of CAN-H(igh) and CAN-L(ow). The two wires have termination

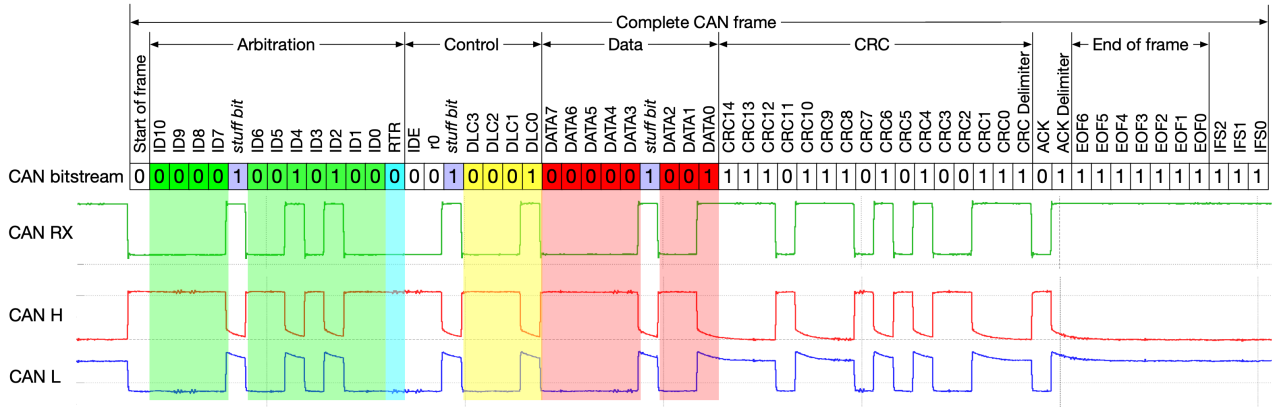



Figure 2.1: **CAN frame with added stuff bits in purple. Ken Tindell, Canis Automotive Labs Ltd, CC BY-SA 4.0** `https://creativecommons.org/licenses/by-sa/4.0`**, via Wikimedia Commons**

resistors at each end to pull the signals to $\Delta 0\,V$. This is the recessive state (binary 1) and is not actively powered. When CAN-H is actively pulled high, and CAN-L is actively pulled low by a controller ($\Delta 2V$, dominant state), this is received as a binary 0.

The CAN bus operates on a set baud rate, and each node must communicate at the same speed. This is often 250 kilobit per second (kbps), 500 kbps or maximum 1 megabit per second (Mbps). To make sure that timing errors do not cause problems, bit stuffing is added. For every 5 bits that are the same, an extra bit is added, which is the inverse. At the receiving nodes, this extra bit is removed, and the extra signal flank is used to synchronise the clocks. This way, all the nodes will stay in sync and not count more or fewer bits than are sent. The downside is that a CAN message can be a different size and take a little longer to transfer, up to 24 extra bits [28].

The CAN bus works best if it is routed like a single-line network. Star topologies and trees should be avoided. It is possible to have drop lines branching off the main line. These should not be too long, and the total network should not be too long (depending on baud-rate) to prevent propagation delays giving timing problems between multiple nodes sending at the same time[26].

The CAN bus is a message-oriented bus where each message is broadcast to all nodes in the network. Each message has an ID of 11 bits (CAN 2.0A) or an extended ID of 29 bits (CAN 2.0B). A node can often receive both types of message IDs. The message IDs identify what data is being sent and also set the ordering of the messages. Messages with a lower ID have precedence over a higher ID. This is done with carrier sense multiple access (Carrier Sense Multiple Access (CSMA)) with non-destructive arbitration[29]. This is done by sending the message ID over the CAN bus and also sensing the current state of the CAN bus. If two nodes try to send a message at the same time, then they will start with sending the ID bit by bit, Most Significant Bit (MSB) first. If the bit value is the same, then the nodes will sense the same value as they send. If node A has an ID with a 1 (recessive) and node B's ID has a 0 (dominant), then node B will sense the dominant, actively pulled value and continue. Node A, on the other

hand, will also sense the dominant bit while it tries to send a recessive bit. This means that node A lost the arbitration phase and should stop sending. Node A can try in the next slot. This method ensures collision avoidance without wasting time slots (non-destructive), meaning that if messages are waiting to be sent, every time slot, the message with the lowest ID will be broadcast.

Normal IDs have precedence over extended IDs. The message ID priority means that more important messages should have a lower ID than less important messages. After the arbitration phase and one node can continue sending, it will send the length of the data. For the CAN bus, this can be 0-8 bytes. Then the data is sent. After the data, there is a CRC value to let the receiving nodes check the data bytes. If the data is received correctly, all nodes should send a dominant bit to acknowledge the correct reception of the message in the single ACK bit slot. This will let the sending node know that it is done and can remove the message from its sending queue. When a node does not receive an acknowledgement, then it can retry and stop after multiple failed tries. The acknowledgement does not indicate that the receiving nodes do anything with the data, just that the transmission was successful to at least one node.

Figure 2.1 shows a single CAN message without and with bit stuffing, the purple bits. It has an 11-bit ID, 0x14 (green bits), with 1 data byte, indicated by the yellow bits. If it is a 29-bit ID, then the IDE bit (identifier extension) would be 1, recessive. This also makes it so that 11-bit identifiers have precedence over 29-bit IDs, as the IDE bit is dominant with an 11-bit message ID. The data is sent after the length, red. The CRC check bits are sent afterwards, and finally, the ACK slot is there for the other nodes in the network to acknowledge the message.

After receiving a message, a node can decide to process/use it or just discard it. Often a node has a bitmask filter to quickly check if it wants to receive this message. Messages do not encode what kind of information they have, like XML or JSON, but the receiving and sending node need to have common knowledge of how to encode the information. This is often done with a DBC file. This file has a description of the whole network, each message, and what signals they contain. The signals each have a type, like int, unsigned int, or float, and have a location in the message, described by start bit and length. Signals can use multiple message bytes, and then the byte order is also important. The two variants are Motorola and Intel. Motorola is big-endian, where the most important byte is sent first. Intel is little-endian and sends the least important byte first[21]. Multiple signals can also be put in a single byte. This way, multiple types of signals can be encoded in a message, and there is less overhead on the CAN network if all bits are used for signals.[6]

64 bits is sometimes not enough data to be communicated. Higher-layer protocols were created to extend this limit. In commercial vehicles like trucks, tractors, and heavy-duty vehicles, the J1939 transport protocol is used for sending up to 1785 bytes in a single 'message'[8].

### 2.1.2 CAN FD

After 26 years with the normal CAN bus, introduced in 1986, car manufacturers wanted to send data with higher rates and send more data at a time. This became Controller Area Network with Flexible Data-rate (CAN FD), released in 2012, an upgrade over CAN. This new standard (ISO 11898-1:2015) is already

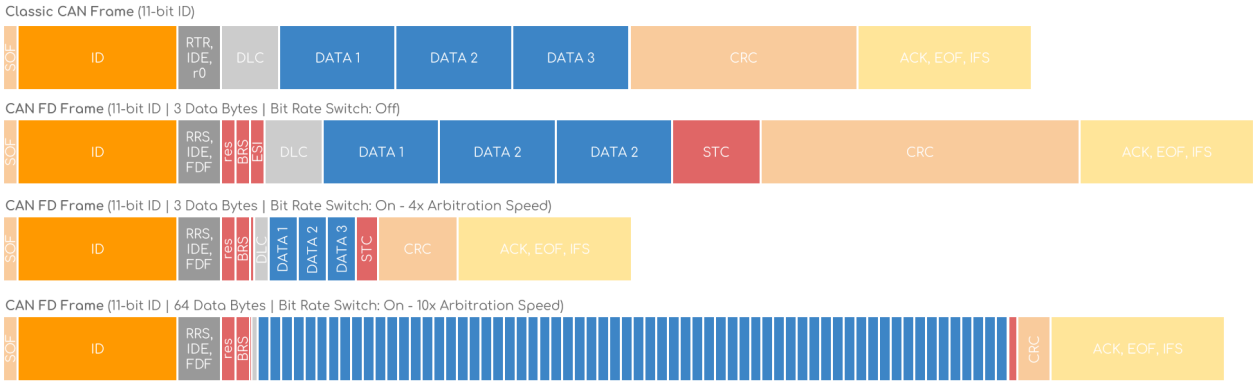



Figure 2.2: **CAN FD frames compared to normal CAN frame.** [7]

in some innovation vehicles and requires gateways to have the needed data on a normal CAN-bus.

CAN FD increases the maximum data size to 64 bytes (8x more than CAN) and adds flexible data-rate. The flexible data rate is only applicable to the (64 bytes) data part of the CAN FD message. A sending node can select to enable or disable the faster data rate. Like normal CAN, the arbitration phase, the first part of each CAN message, is done at a maximum of 1 Mbps. This is to make sure there are no problems with signal propagation through the network. After a single node wins the arbitration phase (depending on the message ID), it can switch to a higher baud rate by setting the bit rate switch. Then it can send the data at a higher baud rate. This can be done because there is just a single node transmitting on the system, and it does not have to listen to other nodes on the bus. Therefore, there is no signal propagation to worry about. The higher baud rate must be configured beforehand on each node and must be the same. [11][2]

The data rate can go up to 8 Mbps but is often set to 5 Mbps. Besides the increased efficiency, also the error checking was improved in comparison to CAN: the Cyclic Redundancy Check (CRC) length is 17 or 21 bits depending on the data length instead of 15 CRC bits in normal CAN. These bits can be sent at the higher rate, so the time impact is less.[7]

Figure 2.2 shows a normal CAN message compared to a CAN FD frame with the same data length and disabled bit rate switch. This frame takes a bit longer to transfer on the network, but when the BRS is enabled with a four times higher data rate (3 $^{rd}$ frame), then the time it takes on the bus is much shorter. The last frame shows a message with a ten times higher data rate compared to the arbitration rate and 64 data bytes. This frame takes a little longer to transfer compared to a normal CAN frame with 3 bytes but carries more than 20 times more data. A 10x difference is possible, as normal CAN usually runs at 500 kbps and CAN FD can run up to 8 Mbps.

CAN FD controllers are backwards compatible with Classical CAN, so a node with a CAN FD controller can be connected to a CAN network. A CAN controller is not forward compatible with CAN FD, so a mix of Classic CAN and CAN FD nodes must all fall back to CAN.

According to Reindl et al., CAN FD is 60% faster than CAN when using 8 user data bytes and a data rate of 4 Mbps while still using the same order of processing time and energy [18].

### 2.1.3 CAN XL

The latest version of CAN is Controller Area Network Extra Long (CAN XL) and was introduced in 2020. CAN XL has been developed to support an even higher data rate and to fill the gap between CAN FD and automotive ethernet (100BASE-T1). CAN XL is not yet in use in (innovation) vehicles, but the HMI system should still be able to connect with it when it does get included in cars.

CAN XL keeps the success factors of CAN FD and improves it similarly as CAN FD versus CAN. It has the same arbitration phase as CAN at a maximum of 1 Mbps. The data is then sent at a rate from 1Mbps up to 20Mbps. It also increases the possible data size to 2048 bytes to support Ethernet tunnelling over CAN XL. Another difference is that it just uses 11 bits for priority ID and adds a 32-bit acceptance field in the data phase for addressing or message IDs. This way, the arbitration phase is over more quickly while still having a big range of IDs.

CAN (FD) does not have a flag to indicate what higher level protocol is used, like in Ethertype in Ethernet frames indicating IPv4, IPv6, or LLDP, but the data in the message has to indicate that. CAN XL adds an 8 bit service data unit type (SDT) to indicate this. This SDT can indicate multiple different types like normal messaging, CAN (FD) tunnelling, and Ethernet tunnelling.

CAN XL is also backwards compatible with CAN FD, meaning CAN XL nodes can receive and send messages from and to CAN FD nodes using the CAN FD message and CAN XL nodes can send CAN XL messages at a faster bitrate (8Mbps max with mixed networks). This is possible because the CAN FD and CAN XL headers are the same until a CAN FD reserved bit that it expects to be 0, but when it is 1, the CAN FD node will go into a passive state until the next message. CAN XL nodes will detect it is a CAN XL message and switch to CAN XL mode. [19] [12] [20]

### 2.1.4 LIN

While some systems absolutely require the fast speeds of CAN or CAN FD, some other systems do not need such speeds. Those systems include door controls, window controls, heating systems, and other non-safety critical systems. Equipping those systems with CAN controllers would be overkill. Therefore the LIN bus was introduced to have a cheaper alternative that is supported by multiple car manufacturers. The safety innovation truck had a steering wheel with touch sensors for hands-on detection. This system used LIN to communicate, and for the truck, a *Peak-System* LIN to CAN gateway had to be used and configured after a failed attempt with an *MRS* gateway. If the HMI system had had LIN support, it would have saved several days developing, configuring, and debugging.

The LIN bus uses a single wire with a (12 V) signal relative to ground instead of the differential pair of CAN. A network consists of one master and (up to 16) slaves. Figure 2.3 shows a LIN frame. The master always initiates the communication by sending a header with an ID it wants to receive or send. The eight

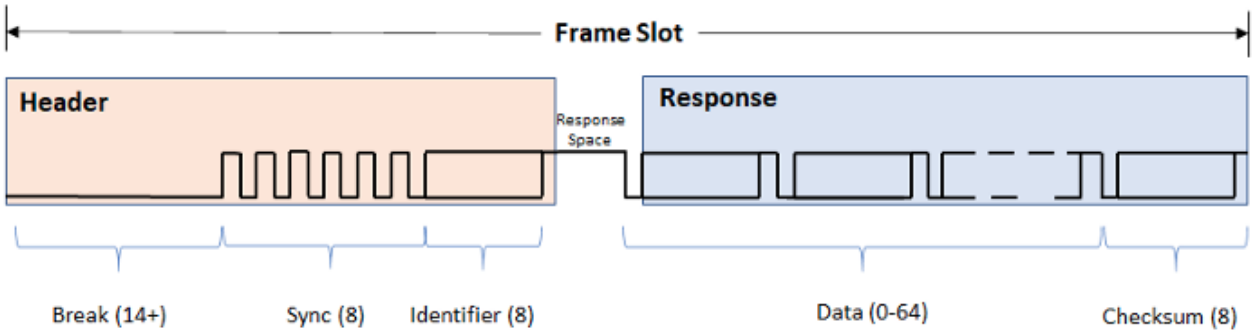



Figure 2.3: **LIN frame.** [4]

sync bits are for syncing up the clock of the slaves with the master. The master or any slave will respond after the header and send the data of up to 8 bytes. Using a single master instead of a multi-master setup like CAN removes the CSMA and arbitration requirement. To have a frequent data interval, the master has a schedule table of when to request or send specific messages. The LIN protocol can be implemented with a Universal Asynchronous Receiver Transmitter (UART) interface, which is often available on microcontrollers. A LIN network can work up to 19.2 kbps. [9] To interface with the rest of the vehicle, the master often also has a CAN connection.

Using a single wire and reusing the UART interface of a microcontroller makes it cheaper than CAN.

### 2.1.5 FlexRay

Another fairly new automotive bus standard is FlexRay. FlexRay (ISO 17458-1:2013) was designed to have better scheduling guarantees. The CAN bus is event-driven with message priority, resulting in the possibility that a lower priority message can be delayed for a (relative) long time. FlexRay solves this problem by having a Time Division Multiple Access (TDMA) scheme. This is a cyclic scheme with a static segment and a dynamic segment. One such scheme with messages is shown in Figure 2.4. The static segment consists of multiple slots, each reserved for an electronic control unit (ECU). Each ECU can have multiple slots. The ECU can then decide to send data in its slot or not, and then it will be an empty slot that cycle. The example shows that on Channel 0, ECU B did not transmit in its slot. The data can be 254 bytes long.

The dynamic segment is for messages that are less critical and can have a longer time delay. The dynamic segment is also broken into slots. Each possible message gets a minislot. This is when the ECU is allowed to start transmission and is thus short compared to the transmission time. If an ECU decides to send a message, then the following minislots are pushed back to after the transmission of the message. The dynamic segment always has the same duration, so messages with a lower priority and, thus, a later minislot will have to wait until the next dynamic segment, where they might be pushed back again. This pushing back is also shown in Figure 2.4. Node D sends message D2 on Channel 0, and thus message C2 is pushed back. On Channel 1, node D did not send a message, and

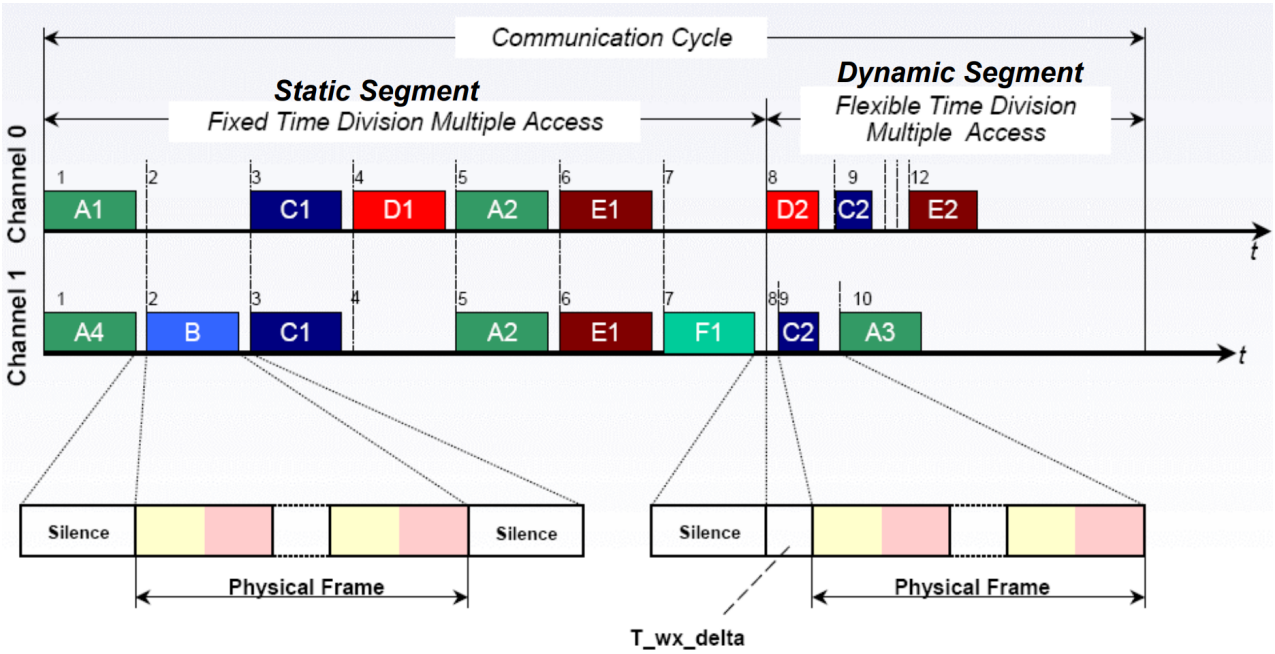



Figure 2.4: **FlexRay cycle. [25]**

thus C2 was sent earlier.

The configuring of the dynamic section and dynamic segment requires more configuration than a CAN bus and cannot be changed on the fly like with CAN bus, where it is possible to add a node and have it working without configuring anything on the other nodes. Each FlexRay node must have the same timing configuration for the segments to have the FlexRay bus working. The FlexRay network can be more diverse than a CAN bus. FlexRay allows for a multi-drop bus like CAN bus, but a star network and a combination (hybrid) network is also allowed. A FlexRay bus uses two twisted pairs, channels A and B. This increases fault-tolerance or increases bandwidth, up to 20 Mbps. Compared to the normal CAN bus, FlexRay offers determinism for message timing, increased data size, and faster transmission, but at the cost of more complexity, and the complete network must be designed and configured. [15] [22]

### 2.1.6 Conclusion

All these buses use different means for collision avoidance and priority, and they differ in speed, data size and other details. However, they are all based around a simple message ID and a message of some bytes. These messages are stateless and cyclic, so message dropping is not a problem. As they are not connection-based, like modern ethernet, but a broadcasting bus, it is easy to pick up data from these buses without interfering with or changing parts of the vehicle.

## 2.2 HMI Communication

The current generation HMI used in the innovation vehicles are all based on a tablet running a web app. This is easier for the developers, as they only have to use the normal browser stack of HTML, CSS, and JavaScript without any other lower-level languages. It is also very easy to develop, as the web app will also run on any computer, tablet, or phone. This way debugging is made very easy, as debugging a web app is easier than debugging a compiled app installed on a tablet.

Using a browser-based HMI also imposes some drawbacks. Not all protocols are allowed to be used, as security is very important. This is needed because the browser will (often) run all the JavaScript present in a webpage. The three protocols to request data that are allowed are HTTP requests, WebSockets, and WebRTC.

### HTTP Requests

HTTP requests are the same requests that browsers issue when loading a page or requesting an image. HTTP requests have a high overhead per request and therefore are quite slow. Each message to and from the server requires a new request as the connection cannot be kept open. A server is also not able to send data without that data being requested or waited on by a browser.

There are two methods to get frequent updates from a server with HTTP requests. The first method is polling. This method repeatedly starts a connection to check if there is an update and the server responds immediately with the data or an empty response. This method has a high overhead and is not very efficient if the data rate is lower than the polling interval. The other method is long polling. Here the server does not respond with an empty message if there is no data, but it waits until there is data or a timeout passes. This reduces the number of HTTP requests and lowers the overhead, but requires the server to keep the connection open and requires the client to restart the request when it receives data.

Figure 2.5 shows HTTP polling on the left. The client only receives data after each request.

### WebSockets

WebSockets are an upgrade over HTTP Requests. A WebSocket connection is started in the same way as an HTTP request, but then is switched to the WebSocket protocol when the browser requests it and the server accepts the upgrade. A WebSocket connection allows for full duplex communication without it being requested from the browser or the server. The server can thus send data when it wants to, independent of what the client sends. This protocol is, therefore, beneficial for systems with frequent updates. It is firewall and proxy friendly, like HTTP requests, and does not require changes on intermediate devices because it uses the normal HTTP protocol and ports. WebSockets use less bandwidth than (polling) HTTP requests and use less memory on the client[23]. The latency is also less than the HTTP request methods[16].

Figure 2.5 shows a WebSocket connection on the right. A connection is once opened and data can flow anytime from server to client and vice-versa.

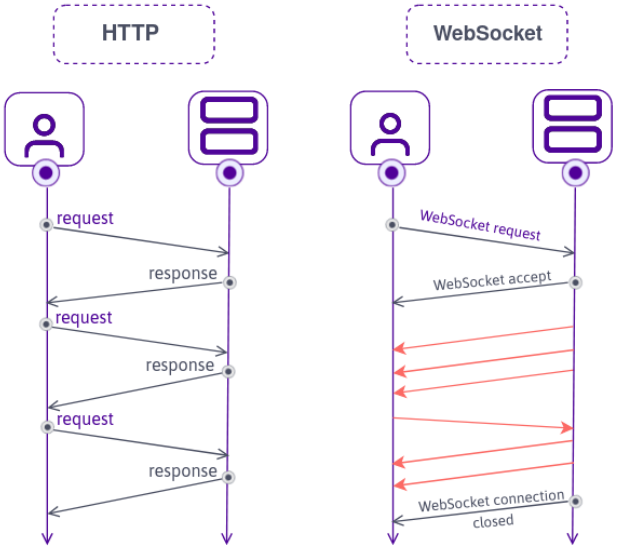



Figure 2.5: **HTTP Requests vs WebSocket [14]**

The current HMI uses a WebSocket connection to the EtherCAN/2 gateway to send CAN settings and send/receive CAN messages. More information on the *ESD* EtherCAN/2 will be given in Section 3.2.

### WebRTC

Web Real-Time Communication (WebRTC) is a set of APIs and protocols for browsers to communicate and stream video, audio, files, or any other data peer-to-peer. It uses a server to detect the IP address and port that the clients might be reachable on. Then two browsers try to connect to each other using this information. This is needed to bypass NAT and firewalls. If direct communication is not possible, WebRTC will fall back to using an intermediate server to communicate. When a communication channel is created, the peers will indicate what protocols they support and what kind of data they want to send. By negotiating a common supported set of protocols, a video or data stream is established and a peer-to-peer connection is set up.[24] As WebRTC is used for audio and video with peer-to-peer communication, it is not usable for server-client communication and thus not usable for the HMI system. It could be used for inter-HMI communication.

WebRTC is often used for online video calls like Jitsi or MS Teams.

## 2.3 Virtualization

Virtualisation could be used for running applications on hardware or operating systems that the applications were not developed or compiled for. The routers in use (MicroTik HapAC$^2$) support virtualisation with Docker containers, so they could serve as a WebSocket server and connect to the hardware interfaces

on the other end. This would lower the number of components, price, and space needed for an HMI system. Besides acting as a gateway, the router could also be used to run a web server to fulfil requirement 2b, serving HMI and debug pages.

Docker is an ecosystem for container virtualisation. It is essentially a system for running code in a virtual machine (VM) without all the overhead of a VM, like the operating system, hardware emulation, and scheduling. Running an application in Docker versus a VM can improve the speed up to 26 times[1].

Docker containers can also be used for the development work of software gateways. Some embedded Linux hardware, like the ZF ProConnect, require a different toolchain with some limitations, and a ready-made container would decrease the setup time for the next system developer.

## Containers

A container is a running instance of an image. These images contain a base layer of an operating system with layers of file changes like software packages and code. This layered copy-on-write file-system approach improves start time and cachablility of intermediate layers, resulting in lower build times[5]. These images can be shared or uploaded to image registries like Docker Hub and then run on other machines. An image has a specific CPU architecture, like x86_64 or ARMv7. An image can only run on the architecture it was built for, except if an emulator like QEMU is used. It is possible to cross-build an image for another architecture than the machine it is built on. It also has to use an emulator for that. An image can run any Linux flavour without any restrictions from the machine's Linux flavour.

Docker is mostly used on Linux operating systems or Windows Subsystem for Linux (WSL). Docker is able to run Windows containers on Windows hosts, but that is not used that often and requires software licenses.

A container is isolated from other containers, so they will not interfere with each other. They can have access to a network, so containers are able to communicate in that way. Using this isolation, it is also possible to run multiple containers with the same image to have load-balancing for other distributed systems. It uses Linux concepts like namespaces and cgroups to make the containers run in isolation and cannot interfere with each other or the main system. For the code in a container, it seems like it is running on real hardware without the Docker layer in between. Using containers makes it much easier to make a portable system that can be run immediately without installing extra packages.[1]

Building an image is often done using a Dockerfile. This file contains the base image name, like Ubuntu, Alpine, a lightweight image, or GCC, for compiling code. After the image, multiple copy and run commands can be specified for copying files from the host to the container and running commands in the container. Each of these commands creates a new layer that describes the changed files in that layer. The consequence of this layered copy-on-write approach is that every layer adds size to the image, even removing files. This can be reduced by squashing the final image, making it into a single layer.

Figure 2.6 shows this layered filesystem approach. Each addition is marked with an A, each delete with D, and changes with C. Only the files that are added or changed when looking from the top are visible to the container, and

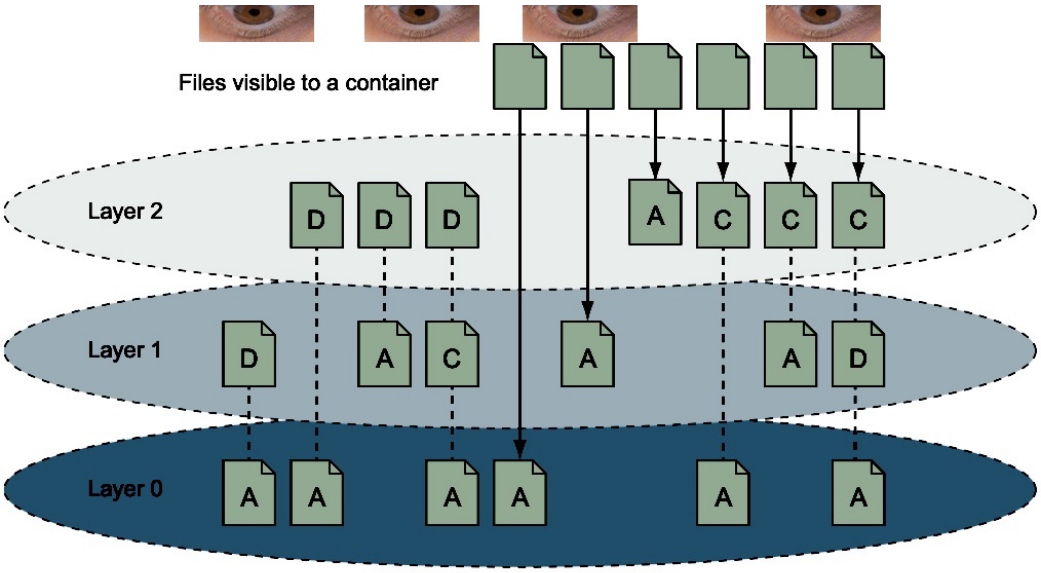



Figure 2.6: **Copy On Write Layers for Containers [10]**

it will take the latest version, the file version in the latest layer it was added or changed.

A better approach to minimising final images is to use a multi-stage Dockerfile. This uses multiple images, where only the final image is exported. For compiling a C++ application, a lot of libraries and a compiler are required, but for running that application, fewer files are needed. A multi-stage build can help, where the first stage builds the application and the second stage is a slimmed-down image where the executable with some required libraries is copied to. The resulting image then does not include the intermediate files and the huge compiler image.

## Usage

Docker is often used in software development for Continuous Integration / Continuous Delivery (CI/CD), where each change of the code is checked, compiled, tested, and (possibly) released to quickly and easily check that the changes do not introduce bugs and that the code still works. Gitlab CI/CD can use Docker as a runner, a system to run the commands in the $.gitlab-ci.yml$ file, and includes a container registry to store images that are generated by and for the CI/CD pipeline.

For software development, Docker is also advised to easily set up a reproducible complex development environment. The benefit of using containers is that a developer does not have to install the tools on their machine, and it makes sure that every developer on a team has the same version of the tools and has packages and libraries in the same locations. Using containers makes it easier to switch to a newer/older version without messing up their original system.

When a system, like a website backend or a database, is developed, then it must be published or put into operation (DevOps). Docker can also help with this. This is often done together with Kubernetes, where the containerd

systems are often used. Containerd is a subsystem of Docker that is the actual container runtime. Containerd used to belong to Docker, but it is made into an independent system.

Kubernetes manages containerised workloads and scales the number of containers based on workload or other requirements. A Kubernetes system consist of a cluster and a set of pods. A cluster is a set of machines (nodes) that can run the containers. There is one master node that schedules and maintains the cluster. The other nodes will run the containers. When there is a high load, the master node will start more containers. A pod is a set of containers that work together, but often it is a single container. A pod can be started on multiple nodes for load-balancing. A website can, for example, have a database pod for storing data and an Apache pod for serving web pages. When there is a high load for static files, the Apache pod can be run multiple times concurrently, but the database pod does not have to. When there is a high load for dynamic files, then the database pod also has to be scaled up.

The MikroTik routers currently in use also support Docker containers since RouterOS 7.5. This can be used to run any (lightweight) container on the router. Other Mikrotik routers like the RB5009UPr+S+IN have more memory and storage to run bigger applications.

## 2.4 Existing messaging systems

The HMI system should essentially be a message broker: sending data from a CAN (FD) bus to an HMI and vice-versa. There are, of course, other message brokers and gateway systems in use to convert signals, interfaces, and protocols to any other type. Inspiration for architectures and components can be taken from them, and the HMI system could even interact with them when required. Below are some existing systems and architectures from previous work.

### 2.4.1 Industrial interconnection

Jian Zongmin et al. designed a software-defined gateway for industrial interconnection [13]. Their goal was to connect multiple industrial devices, like Programmable Logic Controllers (PLCs), in a plant with different protocols. They designed and implemented a gateway running on an ARM Cortex-A8 processor with embedded Linux. They designed the system where a configuration file selects a driver instance for each connected device depending on the application's needs. Then the in-memory database stores the connection information and data. This data is then sent to other drivers when they need information from the database and to an OPC Unified Architecture (OPC UA) service. OPC UA is an industrial automation interoperability standard developed to abstract away the PLC-specific protocols, like Modbus. Another connection to the in-memory database is made to an MQTT (Subsection 2.4.2) service to connect to IoT devices.

The industrial interconnection system is mostly designed to connect all devices to all devices, where the vehicle gateway should connect the HMI to the vehicle buses, where the vehicle buses do not get data from other buses. However, it shows how configuration could be used to select drivers and build a flexible system capable of connecting to different buses or subsystems.

### 2.4.2 MQTT

MQTT is often used in Internet of Things (IoT) systems to communicate sensor data to a central point. MQTT uses a publisher/subscriber model to send messages. It needs a broker that each client connects to over the internet using TCP. The clients then can subscribe to a topic and send messages on a topic. The broker then forwards the messages to each subscriber. MQTT is optimised for resource-constrained devices with possibly limited networking and has Quality of Service (QoS), where the publisher defines the service level. The service level can be 0 (at most once), 1 (at least once), or 2 (exactly once). MQTT messages are all binary messages, with a very small overhead per message.

MQTT could be used for the HMI system, as it is possible to use an MQTT client in the browser. However, it has a lot of features that it does not need, like QoS, and it requires a broker. This broker would then only route all messages from an application with the hardware interface drivers to the HMI, as there are no hardware interfaces with acceptable MQTT support. A system without an extra broker would be more efficient, resulting in no MQTT. MQTT could be added to communicate with other systems with MQTT, like IoT devices, when that need exists, but that is out of the scope of this project and not needed for the HMI system at the moment.

### 2.4.3 ROS

Robot Operating System (ROS) is also a messaging middleware like MQTT, but targeting more towards robotics and less resource-constrained devices. A ROS system is often on a single internal network, contrary to MQTT, where clients can be connected from anywhere, given the broker is accessible. ROS also has the notion of a main controlling unit, the ROS master. This master listens for nodes (the clients) connecting to it and will keep track of which topics it subscribes to and which topics it publishes. When another node joins the system and subscribes to published topics, the ROS master will let the publishing node know about the subscriber, and these nodes will set up a peer-to-peer connection for their common topics. The ROS master, therefore, never receives the messages. This makes the master slimmer than if it would route the messages. The nodes, however, have to do the networking, resulting in a higher load on the nodes and more traffic on some routes, and it does not work great with firewalls and NAT routers.

ROS has support for CAN (FD) with some packages like socketcan_interface, and other hardware interfaces could be connected to with small nodes bridging between ROS and the hardware driver.

Connecting to ROS from a browser is also possible using roslibjs and rosbridge, which has a WebSocket server.

Using ROS would be possible and would make a flexible system that would be able to connect to other systems as well, but the overhead of the ROS master, which only runs on Ubuntu and Windows x86, removing the possibility to use the routers or other hardware, like the ZF ProConnect, to host the gateway functionality.

Besides message passing and coordination, ROS also supports packaging nodes, distributing those nodes, and simulating robots. It is more like an ecosystem than just a package to include for sending messages.

### 2.4.4 Comparing and possible usage

MQTT and ROS are great pieces of software to connect a flexible and diverse network of clients over Ethernet. They do not work over other networks, like CAN or FlexRay, without extra software on both ends. Stefan Profanter et al. looked into these two middleware systems. Their research shows that MQTT generates the least amount of network traffic for setting up a connection and sending data, showing that it is really meant for network-constrained devices. ROS needs the most data for a node to setup a connection to the ROS master, but when connecting is done, the size of the messages is almost as small as MQTT, only 3 bytes more (5 vs 8 bytes overhead) on a message of up to 10.000 bytes. The computing overhead of ROS and MQTT are both high on average. [17]

Messages could be sent using a ROS to CAN gateway, and then the HMI could connect with a JavaScript gateway. Having a topic for each message ID would make the topic list huge and would create a lot of TCP connections. Also, using MQTT and ROS on a gateway would be overkill and not beneficial, as the (global) architecture is always the same, the amount of which buses it is connected to and the amount of HMIs. Converting each binary message to another message, sending it to another process on the same machine, and then unpacking it and sending it over a WebSocket connection is a lot more overhead than just converting a received message to a WebSocket packet. There are also no hardware gateways supporting ROS, and the one hardware series found that supports MQTT, the Ixxat CAN@net NT devices, only allow for a maximum of 128 selected message identifiers. This solution would not work for a diverse system where any received message could be required on the HMI. Therefore using ROS or MQTT does not solve the problem, and these systems cannot be used for the main message passing. They could be used for sending data between software gateways or other systems, but that is out of this project's scope.

## 2.5 Conclusion

One of the main questions of this project is how to take all the different networks and buses into a single usable type without too much overhead. As shown in the previous sections, CAN, CAN FD, LIN, FlexRay, and other vehicle buses work based on a shared medium with messages with an ID and some payload bytes that are broadcasted to all the nodes in the network. These messages are all connection-less and often are cyclic, sending the same data over and over. Therefore it is possible to build a message definition where all the message types fit in, and it is acceptable to drop or delay messages for optimisation purposes. ROS and MQTT are interesting technologies to connect multiple computers together and have a standardised messaging system. As this project often only has a single general-purpose computer and multiple HMIs that are not able to connect to those systems, those systems are not usable for the main communication. To communicate with the webpages running on the tablets, WebSocket is the fastest and best solution, as it opens a full-duplex communication channel with the least overhead, lowest latency and lowest memory usage. It is possible to use Docker containers on the routers to run code, like a software gateway

or web server. Besides that, Docker can also be used to distribute developer images to easily distribute a working set of tools.

# Chapter 3

# Current system

The current system consists of an *ESD* EtherCAN/2 gateway that has a CAN and an Ethernet connection. The gateway is connected to the router in the vehicle. This router has Ethernet and WiFi for the tablet to connect to and the router (often) also connects to a WiFi network outside the vehicle. This way, the developers can also connect to the gateway from outside to develop, test or show the HMI. This setup is shown in Figure 3.1.

The MicroTik HapAC$^2$ is used in almost all projects as this is a very configurable and complete router that can be scripted.

## 3.1 Human Machine Interface

The HMI is running on a tablet as a webpage in a full-screen browser. The tablet connects to the router via Wi-Fi, and then the webpage can connect to the *ESD* EtherCAN/2 using a WebSocket connection. The data received from this CAN gateway is the same data sent on the bus without parsing or decoding. The HMI does the decoding based on a CAN database (DBC) for the bus it is connected to and will then use that data to visualise what is happening in the vehicle. The HMI can also send data to the EtherCAN/2 to control devices on the CAN-bus.

Decoding on the HMI results in a lower network overhead, makes it possible to change the hardware interfaces and gateway easily, and does not strain the gateway.

## 3.2 Shortcomings and benefits

The current gateway is the EtherCAN/2 from ESD. This gateway only supports CAN, so no CAN-FD or any other bus. The gateway has an Ethernet port and a socket interface running on that with the *ESD* ELLSI protocol, a proprietary binary protocol. This protocol is also supported on a WebSocket interface, running on the same device currently used for the HMIs. The protocol allows any webpage/application to connect to it, set the CAN baud rate and some broad filters. The EtherCAN/2 allows multiple concurrent connections, allowing for multiple connected HMIs. This way, no other hardware is needed besides the tablet, router and EtherCAN/2.

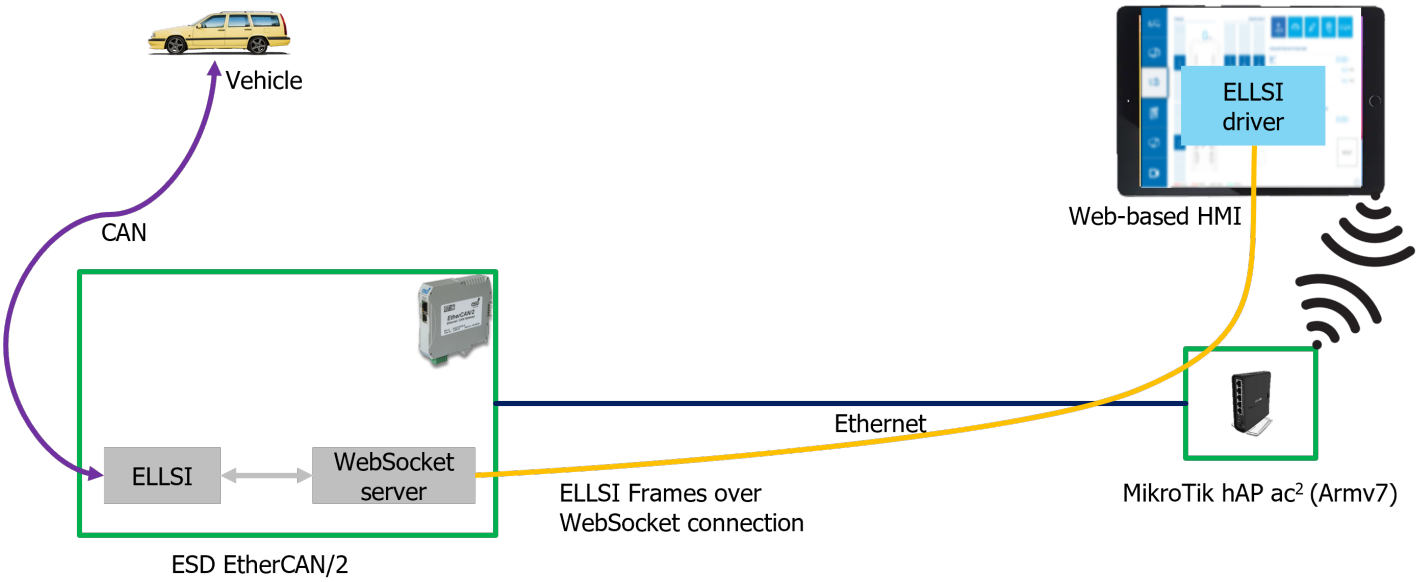



Figure 3.1: **Current system with an EtherCAN/2, router and a single HMI.**

The EtherCAN/2 does not store the CAN configuration, so the HMI has to send some commands to set up the baud rate and (basic) filters. This makes it easy to swap hardware as only the webpage has the configuration and will upload it at every start. And when a setting needs to be changed, it is done in only one spot, and a reload of the HMI will push them to the hardware interfaces.

The data is sent over a WebSocket connection as binary data and decoded on the HMI. Each WebSocket message has a little overhead of a few bytes(2-14) in addition to the lower levels network stack overhead(TCP). Sending each CAN message in a separate WebSocket message would result in a high overhead compared to the transferred information. To improve this, the EtherCAN/2 packs multiple CAN messages into a single WebSocket message. This results in lower overhead but adds latency because each CAN message is received in sequence from the network but is sent to the HMI in a single packet. Tests show that the EtherCAN/2 uses $10ms$ time slices for each new WebSocket message.

While packing multiple messages into a single WebSocket message improves the network and hardware load, reducing the number of messages by filtering and rate-limiting gives an even better improvement. The EtherCAN/2 only supports two range filters, resulting in a lot of unwanted messages passing the filter and being sent unnecessarily. A better filter that selects specific message IDs or multiple smaller ranges will take out more unneeded data and would result in a better system.

Some data is sent cyclically every 6 ms, with often the same data as previous messages. Safety systems, for example, send and require such short intervals, like a seatbelt retractor constantly requiring a message that it should not retract. These intervals are too fast for the HMI and create an unnecessary load on all

the systems. A gateway that would limit the rate of the data to the HMI to 50 ms, for example, would take out 7 out of 8 messages, resulting in an 87.5% reduction. The EtherCAN/2 does not have these features, but a future system should have that possibility.

A gateway could use a binary protocol to optimise the data transfer that minimises the binary CAN messages only to send the data bytes that were sent on the bus. The EtherCAN/2 does not use this. A CAN message can be 0-8 bytes, and sending a few bytes less can improve the throughput of the system and, together with message packing, could make it more efficient on the network, as the other hardware has enough CPU power for the additional minimising.

Extra gateways are needed to connect other buses to the EtherCAN/2, bridging the wanted bus and CAN. One example is using a PEAK-System PCAN-LIN gateway to connect a steering wheel with LIN to the HMI. Adding these extra gateways requires more tools and more time to develop and does not improve the stability of the complete setup.

Another big shortcoming of the EtherCAN/2 is that each bus needs a new EtherCAN/2, as it only supports a single CAN-bus. This resulted in the Efficiency Truck having 4 EtherCAN/2s, with each HMI connecting to all 4 of them. Having a hardware gateway with multiple CAN (FD) interfaces would result in fewer connections, less network overhead and a more stable system. Also, connecting to a single gateway instead of multiple gateways would decrease network loads on those gateways as they would only connect to a single-parent gateway.

The device has a web server for small websites(<10MB), so some HMIs can be hosted on that. This is a nice feature. However, the limited size has been an issue on a few innovation vehicles.

It is also not possible to have HMIs talk to each other for mirroring views, for example. This currently has to be done by a separate proxy, requiring extra software and resulting in a less robust HMI.

## 3.3 Conclusion

To conclude, the current HMI system uses tablets with web-based HMIs for the ease-of-use, innovative look and ease of developability. The CAN gateway in use is the *ESD* EtherCAN/2. This device is able to send CAN messages over a WebSocket connection to the webpage.

The benefits of this system are the following:

- Configuration pushed from the webpage
- Message aggregation to pack multiple bus binary messages in a single WebSocket message
- WebSocket connectivity for the website to connect directly
- Webserver to host the HMI
- Decoding of messages on the HMI

The features that the current system lacks and that should be implemented in a future system to improve usability and reduce network load are the following:

- Message packing to remove empty data bytes
- Multiple CAN buses per device
- Other buses besides CAN
- Bundling of data from other EtherCAN/2s into single WebSocket connection.
- Precise enough filters
- Rate-limiting possible
- inter-HMI connectivity

# Chapter 4

# Architecture

As this project consists of multiple software and hardware components, even in different configurations, good consideration had to be made on where to place the components, which components are needed and how the communication should be done. In this chapter, the software components are described in Section 4.1, and then the hardware parts and possible uses are explained in Section 4.2. After that, a software component mapping onto the possible hardware is made in Section 4.4, and, finally, a final architecture is shown in Section 4.5. This architecture is then used for the MVP in the next chapter.

## 4.1 Software components

Without software components, the hardware is useless, and the system will not do what it is supposed to do. With too much software, the system could end up over-engineered and offer more options and functionality than needed. Therefore only software features are added that are currently used in innovation vehicles or would have been used if they were possible.

One of those components that could be interesting but are and were not needed is a ROS connection. It is nice to have the possibility to connect to ROS nodes, but it is not used (much) in vehicles, so the possible use-case is low for the added complexity and continuous support needed. The safety truck had one ROS system in use for an experimental object detection system, but a node with CAN was added to talk to the HMI and the rest of the ECUs. This made it seem like a normal ECU and removed the need for the HMI developers to know about ROS.

The software is split up into two programmable parts. The first part is the HMI running in a browser and can be programmed in JavaScript (JS). There are some safety and connectivity limitations, so another part is needed. This is the Software Gateway. This is a single executable with a WebSocket interface on one end and on the other end has hardware drivers for connecting to the different hardware interfaces. It is envisioned to make it possible to run this gateway on any hardware with an operating system, like Windows computers, Linux devices and systems capable of running Docker. This should make it possible to connect with any hardware interface, even when they have an operating system-specific driver or specific hardware requirements.

The main software components are listed below:

- hardware interface drivers needed to connect to the different hardware interfaces
- WebSocket interface, required to give the webpage an interface to connect to
- communication and configuration needed to make the system a flexible multi-purpose system for multiple types of buses and hardware components
- Downsampling: filter and rate limit, needed to have a lower network and CPU usage on the tablets by removing (duplicate) messages

Another component is decoding, which is needed to get the real data from the binary information by using a DBC file. This should always be done on the HMI to keep the transferred data simple and small. The current system also decodes the messages on the HMI and uses this for message timeout detection. As decoding is out of the scope of this system, this part will not be discussed.

### 4.1.1 Hardware drivers

This system's most crucial part is sending and receiving messages from hardware interfaces with a CAN (FD) controller. As there are multiple types of hardware, as shown in Section 4.2, there are also various ways to interface with them. Some devices have a TCP/UDP protocol over Ethernet, some are connected via USB, and some hardware interfaces can run the Software Gateway and use some internal protocol. Combining all of these protocols into a single implementation is not possible. However, it is possible to generalise them into a common (abstract) interface that each driver can implement. This allows for the addition of a new driver later for new hardware interfaces. Functions like start, stop, set config and send a message should be implemented by each driver. Otherwise, they do not work. Virtual classes/functions can easily be made in JS and C++. These are the selected languages for the MVP. JS is required as the HMI webpage can only run JS, and C++ was chosen as some hardware interfaces only have a C or C++ driver, and C++ is then preferred as it supports objects and classes (object-oriented programming).

Both the HMI and the Software Gateway need an implementation using abstract classes for drivers to adhere to a common abstract interface as shown in Figure 4.1. The HMI as it needs to be able to connect to the Software Gateway and hardware interfaces using WebSockify and to check the configuration of all the possible driver types before uploading. The Software Gateway needs it as it connects to different hardware drivers.

### 4.1.2 WebSocket Interfaces

WebSockets are the main communication method for HMI as this is the fastest full-duplex communication that a webpage can use. This, however, requires the other side to have a WebSocket server. The *ESD* EtherCAN/2 used previously has this feature, but a lot of networked devices do not have this additional interface. Devices that do not have a network interface like Ethernet or Wi-Fi

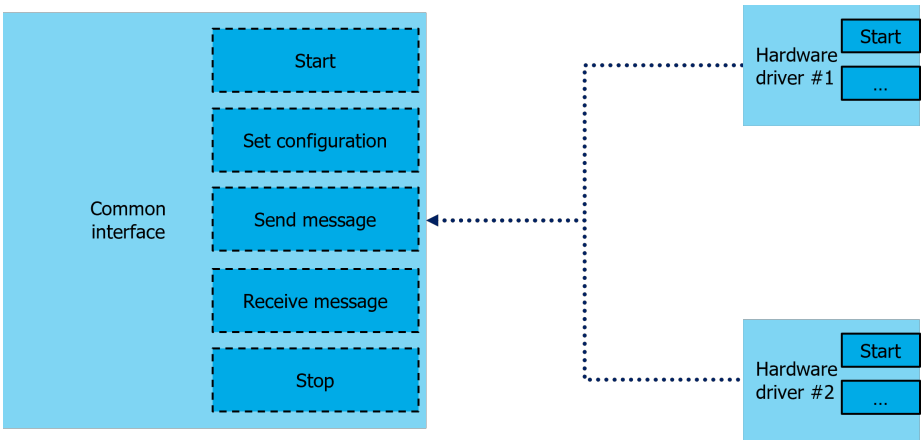



Figure 4.1: **Classes for hardware interface drivers must adhere to a common abstract interface**

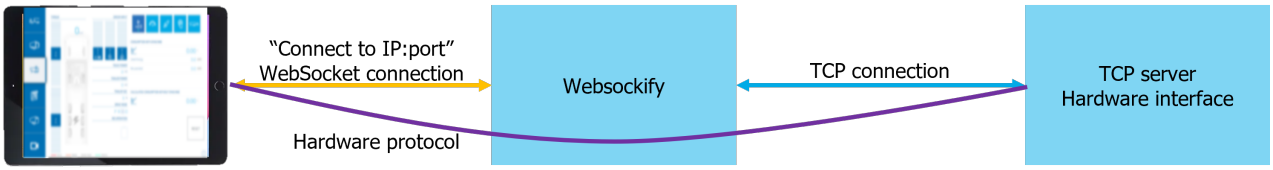



Figure 4.2: **Using WebSockify to connect an HMI to a hardware interface by converting the hardware protocol messages in WebSocket messages to TCP/UDP messages**

cannot be controlled from the HMI, and thus innovation vehicles with those components need a Software Gateway.

### WebSockify

It is not required to use the Software Gateway and added features/complexity if there would be a simple WebSocket to socket converter. Fortunately, there is such a system called WebSockify. This open-source application translates WebSocket messages to socket messages. This is shown in Figure 4.2. The HMI connects to the WebSockify WebSocket server (yellow line) and instructs it to forward all packets to a specific IP and port pair (blue line). Then the hardware interface protocol, which generally does not work from the browser, is run in the browser, making it seem like it is connected directly to the hardware interface (purple line). Each HMI can connect to the WebSockify WebSocket interface and instruct it to connect to a TCP port of any hardware interface. Then the data sent to WebSockify is forwarded to the hardware interface, its Socket interface. WebSockify can be run on the router or any computer in the network.

This makes it easier for the JS developers to add support for a new hardware interface, as they only would have to edit the JS code and no deployed and compiled software needs to be changed. It, however, could result in a higher network load, as each connected HMI needs a WebSocket and a TCP connection

on the network. Using the Software Gateway instead would result in a single TCP connection and a WebSocket connection for each HMI.

For simple systems without inter-HMI communication and only networked hardware interfaces, using WebSockify as the WebSocket interface could be the best option.

#### Websocket interface Software Gateway

The Software Gateway needs a WebSocket server for the HMIs to connect to. There are multiple C++ libraries that provide such functionality, like WebSocket++ and Boost.Beast. It is possible to broadcast data using these libraries to all the connected clients simultaneously and thus treat them as a single connection.

### 4.1.3 HMI Communication

There are two different streams for the HMI communication with the Software Gateway. One is the fast 'cyclic' data from the buses that the Software Gateway is connected to. Besides the message data, there is the configuration stream. This stream of data is often longer and has less priority.

As WebSockets support two payload types, binary and text, each stream can be used on its own type. This makes it easier to parse on the HMI and Software Gateway sides and minimises the network overhead for the bus messages while keeping the possibilities for the configuration data.

#### Bus Messages

The bus messages are the messages in a WebSocket message for each received CAN (FD) message. As these buses run at a high baud rate ($\geq$ 500kbps), a lot of messages can be received per second. The system must be fast enough to process and transfer these messages to the HMI. The HMI can also send messages, however, often at a lower rate.

Sending these messages as binary without processing, like decoding or converting them to JavaScript Object Notation (JSON) text, is the most efficient and the fastest on both ends. As each message can have a different length, ranging from a maximum of 8 bytes (CAN) to 64 for CAN FD to 254 bytes for FlexRay, making each message fit the maximum results in a lot of unused network traffic. Therefore minimising should be used to shrink the message to only the needed size.

In addition, message aggregation should also be used to put multiple bus messages into a single WebSocket message. This reduces the overhead of the WebSocket messages compared to the payload and results in fewer system calls and interruptions on the HMI. This all makes the system more efficient. The downside of message aggregation is that the messages do not arrive simultaneously at the Software Gateway. Therefore, bus messages must be buffered before sending the entire buffer to the HMI. This buffering adds an extra latency depending on the busload, where a higher busload results in lower latency. A timeout has to be added to transfer the non-full buffer after some time to remove very long latencies. Appendix D proposes a message definition and shows an example of buffering and a timeout.

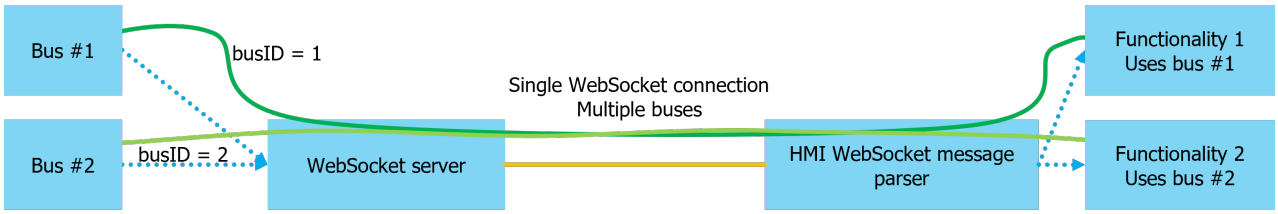



Figure 4.3: **Bus ID message tagging**

As the Software Gateway can connect to multiple hardware interfaces simultaneously, interaction with multiple buses is possible, and bus messages can be tagged by their origin/target bus. This tagging makes it possible to stream all the messages from all buses over a single WebSocket connection instead of creating multiple WebSocket connections. The message aggregation can result in more efficient messaging (buffers are fuller before a timeout) and lower latencies (buffers are full faster). This mechanism is shown in Figure 4.3.

#### Configuration

Configuration is done less frequently and often before needing the faster bus messages. Therefore this can be done less efficiently and more expressively, like sending a JSON text configuration. This allows for sending other low-priority messages, like status messages.

### 4.1.4 Message Downsampling

Some messages are sent very often, and some messages are less interesting for the HMI. These messages should be downsampled by rate limiting or even filtered out. A more aggressive downsampling configuration will result in fewer unused messages being sent from the Software Gateway, resulting in less network traffic and a lower load on the HMI device.

#### Filtering

Filtering is taking out the messages that are not needed based on message ID. There are multiple types of filtering possible.

The most basic version is a range filter as used on the *ESD* EtherCAN/2. This filter has a start and end ID, and every message in that range is accepted. When using a single filter, it is not that useful for an HMI, as the message IDs are often not sequential and span huge ranges, resulting in a lot of accepted messages that should not be.

A more specific filter is a bitmask filter. This is often used in embedded systems, like ESP32 and SocketCAN, as this is cheaper to check than any other filter. It uses a mask and a filter ID. The mask indicates the important bits that must be equal to the bits in the filter ID & mask. This filter works well when the IDs are close together and only differ in a few bits, but for an HMI with large distinct sets of messages, a single bitmask filter will not result in many filtered-out messages. The most specific filter is a list of IDs in which the HMI is interested. This filter will never pass any unwanted messages but is more CPU-

and memory-expensive as each time a message has arrived, the list of IDs has to be looked through to check for a match. This checking could be optimised by sorting and a search algorithm, but it is still more expensive.

For non-extended message IDs, it can be optimised by using a boolean array with a boolean for each possible ID. This optimisation is possible as non-extended IDs are 11 bits, resulting in 2048 possibilities. This range of possibilities would be an acceptable array of 256 bytes, where extended IDs are 29 bits, resulting in an array of at least 67MB. This considerable array would not be a great filter as this uses too much memory.

A mix of filter types should be used for fast and optimal filtering. This way, some bitmask filters could cover almost identical IDs, some range filters for small ranges and a small list of IDs for the outliers. This mix of filters would be faster than a single list of IDs and would filter out more than a range or bitmask filter. The HMI developer must decide what filters to use.

#### Rate limit

Some ECUs send messages at a higher rate than the HMI can ever show. Transferring these superfluous messages creates unwanted network load and should be limited to a more sane interval. For this, a rate limit system should be used that only forwards the same message ID a few times per second.

Rate limiting requires more processing power and memory than filtering, which requires keeping track of messages or the last forwarding time. Therefore, filtering the messages before the rate-limiting step is more efficient. Some algorithms for rate limiting are described in Appendix C.

Rate limiting can decide to discard or forward messages based on the arrival time and does not need not look at the data and whether it is essential or changed. This ignorance is often not a problem, as messages are sent cyclically with a higher frequency than the data changes. Rate limiting based on data requires more memory and processing than deciding on timing, as the data has to be compared, copied and stored, compared to a shorter time variable.

#### Conclusion Downsampling

There are multiple methods to reduce the number of messages forwarded from the buses to the HMIs. Two are filtering by message ID and rate limiting based on time. Filtering can be done by using bitmasks, ranges, lists of IDs or a mix of all of them. The first method (bitmasks) is the fastest but will pass more unwanted messages as the list of wanted messages increases. The last filter (list) is more computationally expensive but will not forward the messages it should not. As mentioned earlier, a combination of the filters will probably be the best option depending on the messages the HMI needs. Rate limiting is more expensive than filtering, and thus rate limiting should be done after filtering.

#### Cyclic broadcasting

Another system that could be beneficial for the HMI system is cyclic broadcasting. This system can reduce the load on HMIs by sending cyclic messages to a specific bus from the Software Gateway instead of letting the HMI do that work. One example of a system that needs cyclic messages is motorised seatbelts. The

retractors need a control message every few milliseconds. Otherwise, they stop retracting for safety purposes. The Software Gateway could take this cyclic sending task from the HMI, resulting in a lower CPU load and a lower network load. There also needs to be a timeout functionality. Otherwise, a button on an HMI might be pressed to let the seatbelts retract, and then the HMI might be disconnected. Some people could end up in continuous tight webbing when there is no timeout check. The timeout would require the HMI to send the message every second to let the Software Gateway know it still wants to send that message cyclically. A message a second is less network and CPU heavy for the HMI than every few milliseconds.

The HMI is also not always able to adhere to strict timing, as the tablet, browser and networking are not real-time. Giving this task to the Software Gateway, with better timing and probably more free processing power, will make the HMIs less loaded and can result in better timing.

SocketCAN has the Broadcast Manager (BCM), capable of sending a message cyclically to CAN (FD) buses. A custom implementation is required when not using SocketCAN. An algorithm was thought of that can send multiple messages at different intervals. This algorithm is shown and explained in Appendix B.

### Conclusion Software components

An HMI system that can connect to multiple hardware interfaces requires some software components. The main components for this project are hardware interface drivers, WebSocket interfaces, communication, and downsampling. Multiple drivers are needed to communicate with the different hardware interfaces that exist. There are multiple kinds of communication, like the Socket interfaces, proprietary drivers and SocketCAN. These all work differently, and a common interface should be made that each hardware driver adheres to with some glue code. This makes it flexible to change hardware and build HMI systems for different innovation vehicles.

Communication with a gateway between the hardware interface and the webpage must be done with WebSocket connections, as this is the only fast communication protocol possible. The protocol in these WebSocket channels must be optimised and scale dynamically with the size of the bus data, and multiple bus messages must be packed into a single WebSocket message. This will reduce the network overhead and will interfere less with other networked systems.

Downsampling is essential to reduce the processing and network load by removing unwanted bus messages as soon as possible. The first stage is filtering by message ID, which can be done with bitmask filters, range filters and plain lists of IDs. After these simple and fast filters, rate limiting can remove duplicate messages in the same short interval. As messages on automotive buses are often cyclic and the same data is sent repeatedly, this does not lead to data loss. Also, the HMI will not be able to display each message at high rates.

Cyclic broadcasting can take some load from the HMIs by cyclically sending data from the Software Gateway instead of letting the HMI do that work. A timeout system is needed to prevent accidents when an HMI is disconnected.

These features will make it a more flexible and efficient networking and processing system.

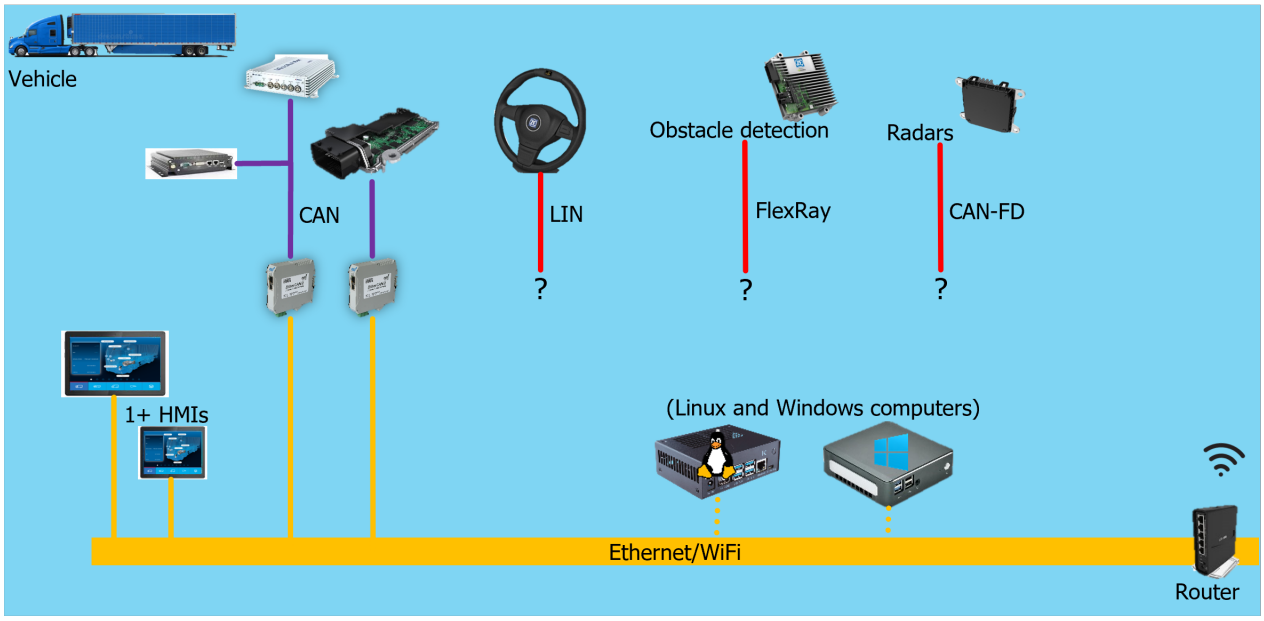



Figure 4.4: **Network layout with default components**

## 4.2 Hardware components

The innovation vehicles need some hardware to run the HMI system. The most obvious component is the tablet running the webpage inside the vehicle. Another essential part is the router that the HMI connects to and then the hardware interfacing the automotive buses. The basic components and network layout is shown in Figure 4.4.

The basic components are listed in Subsection 4.2.1, and options for running software are listed. In Subsection 4.2.2, some hardware interface types are discussed as to how the drivers of those types usually work.

### 4.2.1 Basic components

The basic components that can be used for any processing are these:

- HMI on tablet
- router
- optionally a computer/single board computer

The HMI runs on a tablet, (almost) always an Android tablet, capable of running apps. It is impossible to run Docker containers on Android (or iOS) without root. Rooting the tablets should be avoided, as this is not easy and makes the tablet not stock anymore, resulting in difficulties when swapping them for a fresh one when there are issues. Running anything besides the HMI on a tablet, like a Software Gateway, would not be a great solution. The operating system can throttle or close applications, and the tablets are always

connected by Wi-Fi, resulting in longer latencies and lower bandwidth than a system connected with Ethernet.

The router is used to connect from the outside to the vehicle, connect to other devices in the network, like camera systems, and connect the tablet to the network and hardware interfaces. The router used most often is a MicroTik HapAC$^2$. This router can be set up to have a Wi-Fi network while connecting to an outside network through Wi-Fi, LTE or Ethernet. The router also supports Docker containers that can run any capable application.

Sometimes a computer or a single-board computer (SBC), like an Nvidia Jetson Nano or Raspberry Pi, is added for extra features, like running object detection, GPS location logging or running an extra HMI. These always run Windows or Linux, like Debian or Ubuntu, and can be used for the Software Gateway. As the operating system and CPU architecture are not always the same, a flexible system must be used to have a single code base that works on all these targets. Windows only runs on x86 processors, while Linux can run on a lot more architectures, like ARMv7/8 and RISC-V. Using a general-purpose language like C++ together with a build-system like CMake makes it easier to develop for multiple target systems. CMake is able to select the correct compiler for each build, like MSVC for Windows builds, GCC or Clang for Linux and a cross-compiler for very specific targets, like the ZF ProConnect.

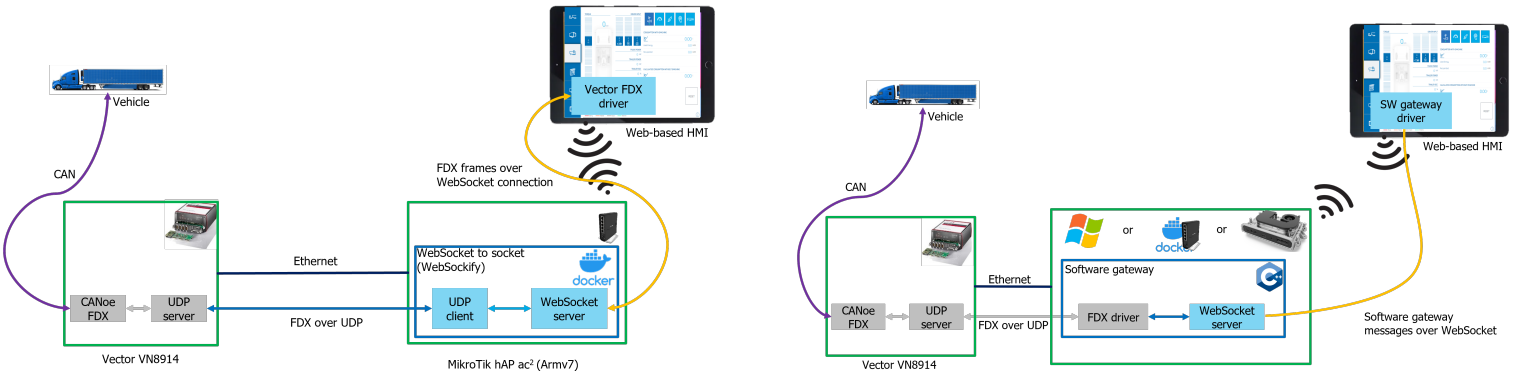
### 4.2.2 Hardware interfaces

As there are multiple buses and manufacturers, there are multiple hardware interfaces interesting for the HMI system. Some only support CAN, whereas others have multiple buses in a single box. For the architecture, the specifics are not that important, but it is important to note that there are a few different kinds that impact the architecture and possibilities of the final system. The main three types are networked, use SocketCAN or use some proprietary driver. These three kinds and their impact on the HMI system are explained in the following subsections.

#### Networked

A lot of hardware interfaces have an Ethernet connection and can be reached from anywhere in the network, but there is no standard protocol for these devices. For example, the *Vector* VN8914 gateway uses the Vector FDX protocol, a binary protocol that works with requests for data, while the Ixxat NT gateways use an ASCII protocol that will just send all the messages. These protocols are often public and sometimes also have a C/C++ implementation. With this, the HMI can connect to them through the Software Gateway or connect to them directly with the WebSockify gateway running on the router, for example. That last route is shown in Figure 4.5a, where a WebSockify lets an HMI connect to the hardware interface directly. The other method is shown in Figure 4.5b, where a single Software Gateway connects to the hardware interface and the HMI connects to the Software Gateway.

Some devices have a limit on the number of concurrent connections possible. This can be avoided by using the Software Gateway or improve WebSockify by combining multiple WebSocket connections into a single TCP/UDP connection.




(a) **Networked hardware interface with WebSockify** (b) **Networked hardware interface with a Software Gateway**

Figure 4.5: **Different methods to connect networked interfaces to an HMI**

These hardware interfaces do not significantly impact the architecture, only that there is a gateway between the HMI and the hardware interface.

### SocketCAN

Linux has CAN (FD) support in the kernel. This is called SocketCAN and works like a normal socket interface used for normal networking, but instead of Ethernet packets, CAN packets are sent. The kernel then invokes the driver specific to the hardware it is running on, and the message is sent. Receiving works the same as reading normal network traffic. SocketCAN also has a system called Broadcast Manager (BCM). This manager can cyclically send messages at specified intervals indefinitely or a few times. This relieves an application wanting to send cyclic messages by offloading this task and results in a lower CPU usage and stricter timing. This broadcasting is an important feature currently performed by the HMIs, taking CPU and network resources while not being strict on timing.

Some hardware vendors sell hardware with CAN (FD) with support for SocketCAN. Using this hardware makes it easier for users to switch hardware without changing any code. The downside of this flexibility is that it only works on Linux with SocketCAN-capable hardware and cannot directly be interfaced from the HMI.

One of those devices is the ZF ProConnect, an automotive computer with CAN and CAN FD interfaces. It runs a Linux distribution and has the required SocketCAN drivers. The Software Gateway could be compiled for this target with a SocketCAN hardware interface driver, and then the HMI could work with this. An architecture with a ZF ProConnect with a Software Gateway running is shown in Figure 4.6.

### Proprietary drivers

The last main type of hardware interface uses proprietary drivers only supported on Windows. These include the Vector XL driver for the *Vector* VN7640 and the Vector CanCases. These drivers are often implemented in C or C++ and

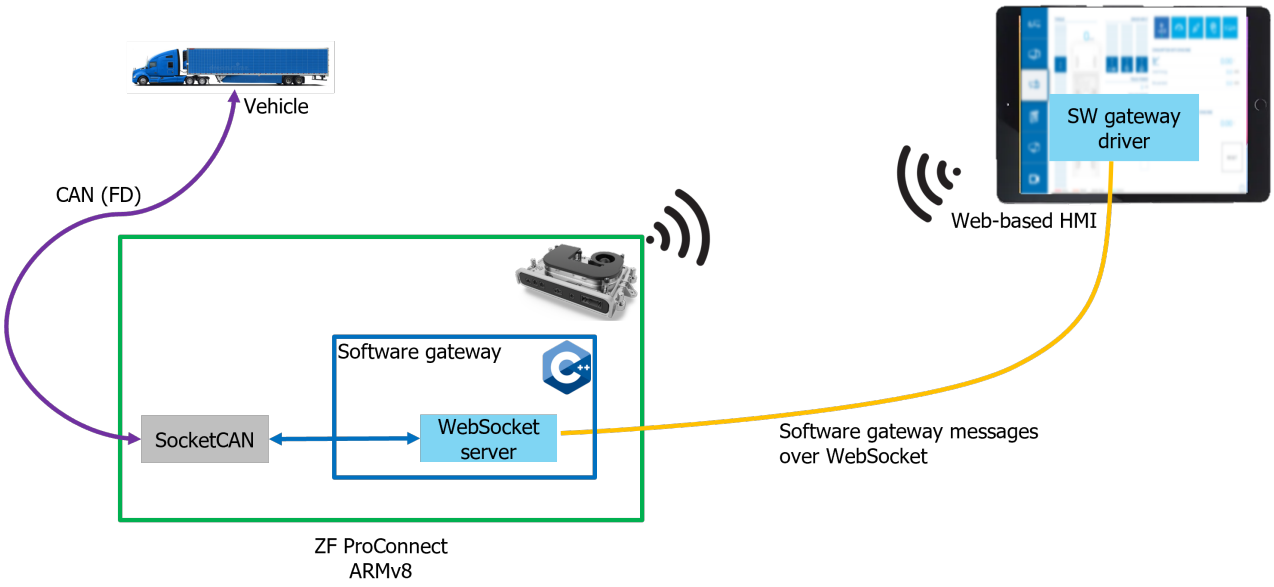



Figure 4.6: **SocketCAN interface with a Software Gateway**

are distributed as a shared library (DLL) and a header file. As these drivers use operating system-specific system calls, it is not possible to run these drivers on Linux without using virtual machines. Some other vendors supply binary drivers for Windows and Linux, but only for x86, so these drivers are incompatible with other CPU architectures.

These hardware interfaces with proprietary drivers impose a significant restriction on the possible systems. They require a specific operating system and architecture and are only usable with the Software Gateway and not WebSockify. One example of a proprietary driver is shown in Figure 4.7, where a *Vector* VN7640 is interfaced from a Software Gateway using the Vector XL driver that only works on Windows.

#### Conclusion hardware interfaces

There are multiple types of hardware interfaces that each have their own protocol and driver. Some devices have an Ethernet connection with a public driver and can be connected from any intermediate gateway, like WebSockify and the Software Gateway. Other devices can run code and be used with SocketCAN and, therefore, can be used from the Software Gateway. The most restrictive devices have a proprietary driver that can only run on a specific operating system or architecture and forces the innovation vehicle to have specific extra hardware.

## 4.3 Distributed systems

In the current system, it is only possible to connect to a single bus per WebSocket connection. HMIs cannot connect to other HMIs or to other vehicles without

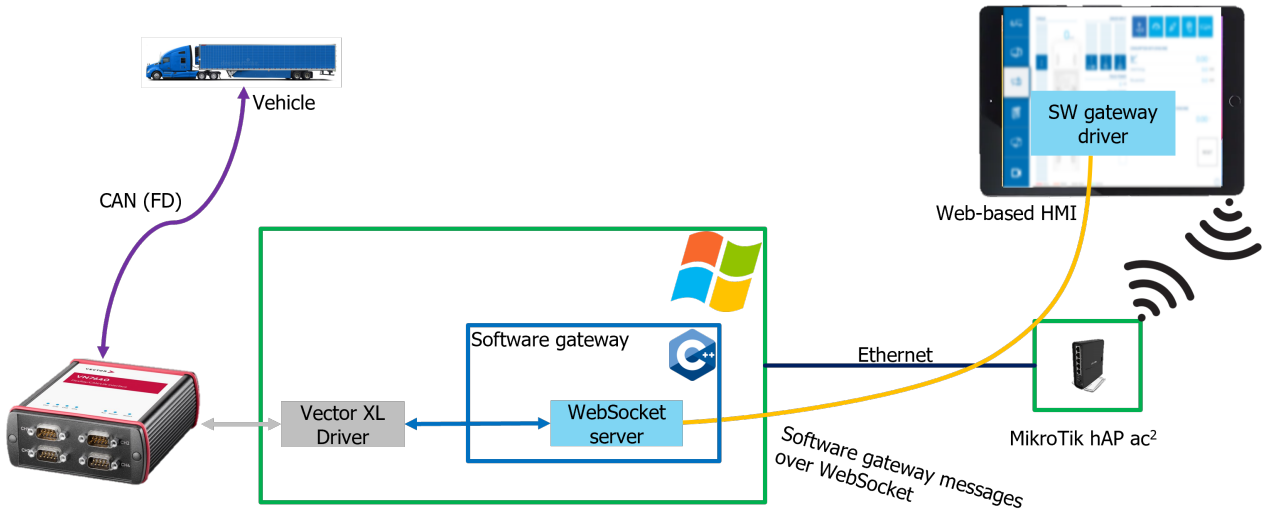



Figure 4.7: **Architecture for the *Vector* VN7640 interface with a proprietary driver for Windows**

using a Virtual Private Network (VPN) to that vehicle. Setting up a VPN connection on each tablet and computer that needs to connect to the vehicles is not recommended, as it needs setting up and also gives access to all the other components connected to the vehicle networks.

Adding components to the system, like special drivers, could solve these problems and improve usability.

### Remote development and viewing

One of those cases where it is impossible to connect to the vehicle network is remote viewing or remote development. In these cases, the HMI is on another network with no possible route to the vehicle network.

A particular driver in the Software Gateway that connects to a remote Software Gateway could solve these issues. An added router, local to the user, could host the Software Gateway and connect through a VPN to the vehicle router, running another Software Gateway. Figure 4.8 shows this solution.

### Mixed systems

It should also be possible that more than one hardware interface is connected to the vehicle. Multiple interfaces could lead to multiple Software Gateways in a vehicle, for example, when a VN7640 and a ZF ProConnect are connected. The VN7640 requires a Windows computer and the ProConnect is also a standalone computer. Then the network should have the data of one gateway be routed through the other Software Gateway, as this leads to less WebSocket connections for the HMIs, fewer WebSocket connections for the child Software Gateway and could lead to better filled WebSocket messages as there is more data to fill the buffer in time. The proposed Software Gateway driver that connects to another

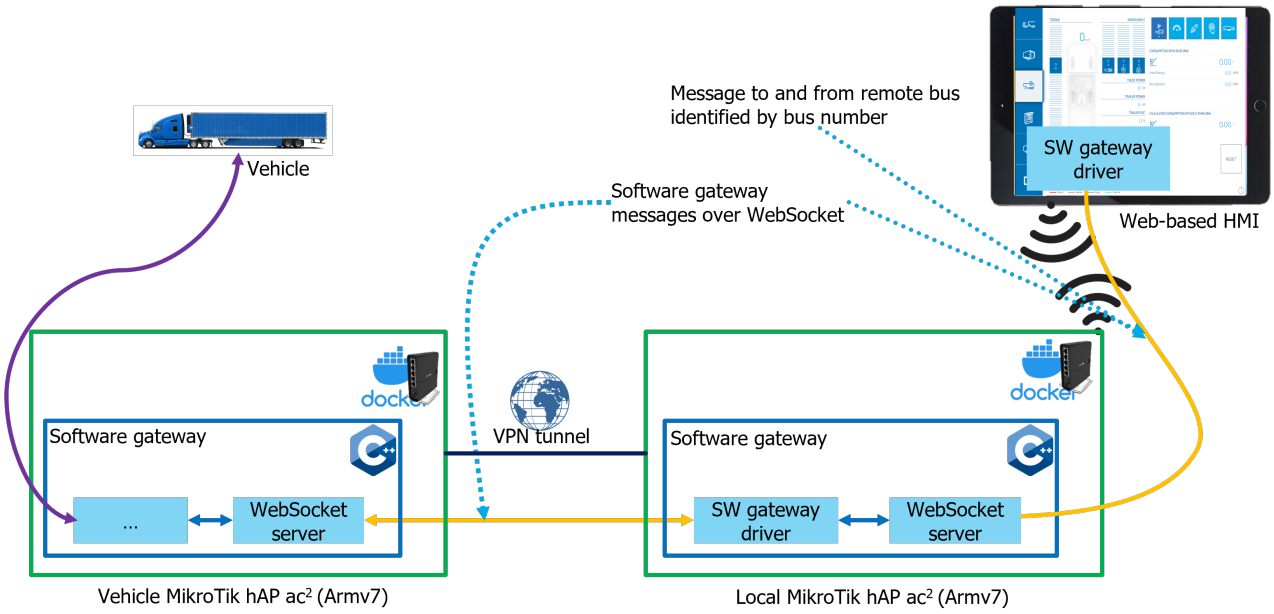



Figure 4.8: **Remote working using a local router connected to a vehicle router**

Software Gateway will solve this problem.

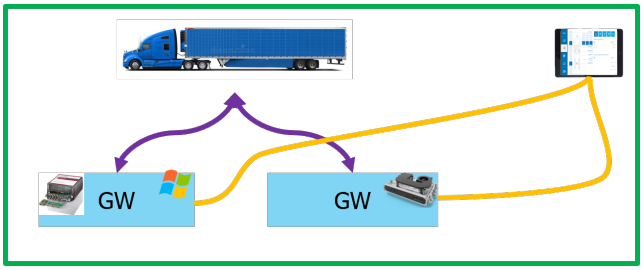
Multiple systems of these mixed systems are shown in Figure 4.9. The first three systems (a-c) show a Software Gateway on a Windows host connected to a Vector XL hardware interface (VN7640). The trucks also have a ZF ProConnect with some buses. Both gateways' data must be sent to the HMI.

Figure 4.9a shows the basic default option, where the HMI connects to both Software Gateways and has two WebSocket connections. Figure 4.9b shows another possibility. Here the second Software Gateway, on the ZF ProConnect, connects to the Software Gateway of the Windows computer. The data of the VN7640 is then routed through the Windows gateway to the ProConnect and then broadcasted to all the connected HMIs. Sending a message to the bus of the VN7640 is the same as sending a message to the ProConnect, albeit with a different bus number. The Software Gateway on the ProConnect will forward it to the second gateway. The gateway on the Windows computer just has one WebSocket client, interacting like an HMI. Routing the data through another Software Gateway, will result in fewer WebSocket connections for the HMI and the child Software Gateway.

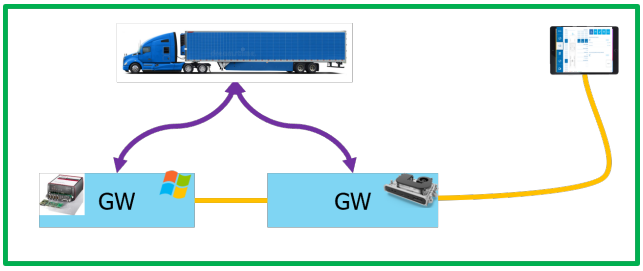
Routing data through multiple Software Gateways can be done as well, as shown in Figure 4.9c, where a router, local to the HMI, connects through a VPN to a gateway in the vehicle. That gateway is also connected to another gateway. This keeps the number of connections for the Software Gateways low.

## Multi-vehicle systems

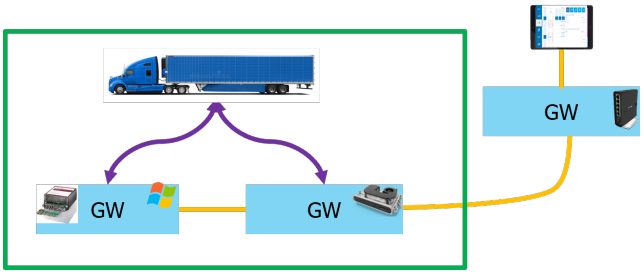
Besides remote viewing, another scenario with VPNs is multi-vehicle systems. One example is showing the locations of the vehicles on a map. The ZF



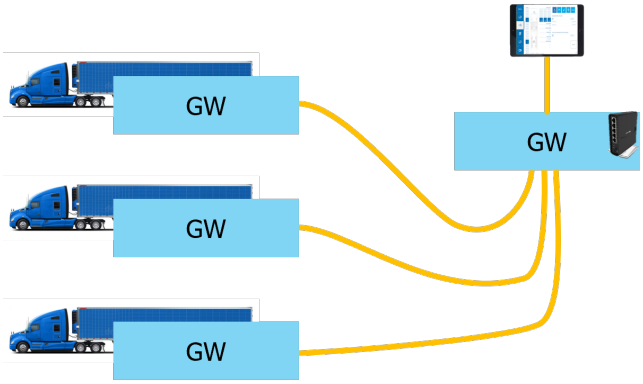

(a) **Default setup with a connection per gateway**




(b) **Routing data from child gateway through parent gateway**




(c) **Remote HMI with data routed through a local Software Gateway on a router to the vehicle**




(d) **Fleet data combining at the router results in a single WebSocket connection per HMI**

Figure 4.9: **Different multi-type architecture**

ProConnect has GPS on-board, so it is possible to develop an application that sends messages with the current location. Connected HMIs could then show that on a map.

This works well with one vehicle, but with remote HMIs and multiple vehicles, this will give problems with the VPNs, as the routers in the vehicles all have a separate VPN. Often a computer or tablet can only connect to a single VPN and not multiple.

One way to solve this issue is by putting a Software Gateway on a central router, as they can connect to multiple VPNs. This central Software Gateway then connect to the separate Software Gateways in the vehicles, as shown in Figure 4.9d, and forward the location messages to the HMI.

## inter-HMI Communication

Sometimes it is required to have inter-HMI communication, for example, when controlling a vehicle system that only accepts non-conflicting data or when pages should be mirrored from one HMI to another.

One example of such a vehicle system is the regenerative braking system in the Efficiency trailer. This trailer has a battery and an electric motor, and with a slider on the HMI, passengers could feel the impact while driving. This system did not allow conflicting data, like one HMI sending a torque value and

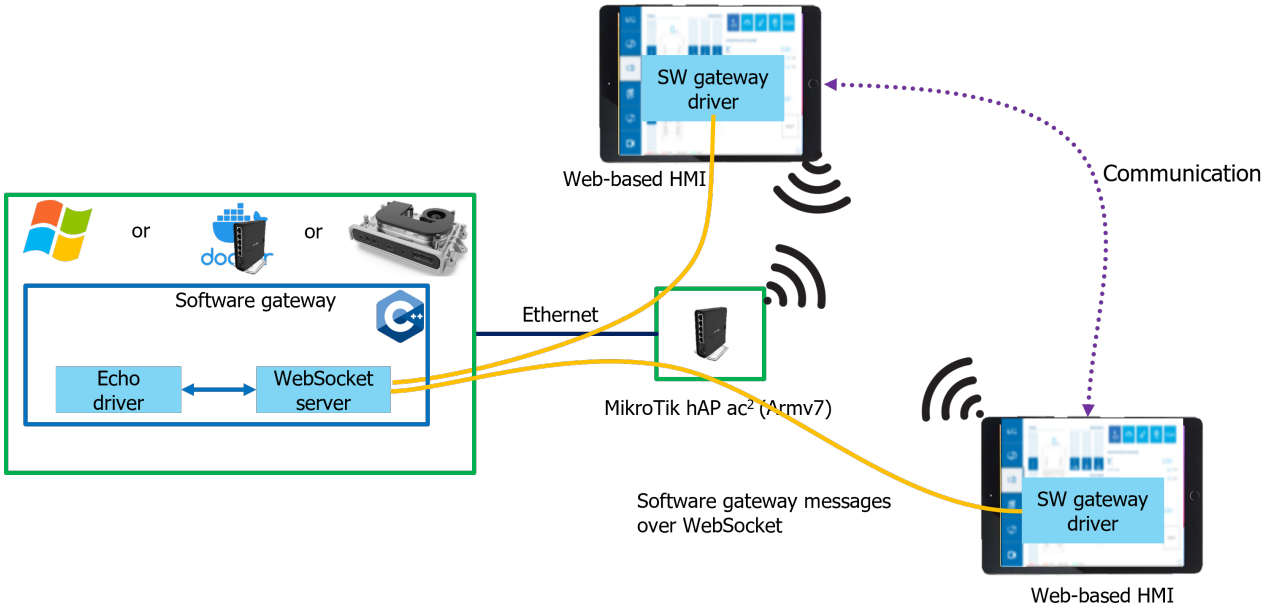



Figure 4.10: **inter-HMI Communication architecture**

the other HMIs sending 0 as a torque. As this HMI system used the current system, no inter-HMI communication was possible, and the developers opted for only allowing a single (predefined) HMI to send data. The other HMIs were not sending data.

The new HMI system could implement a special driver that echoes all the messages sent to it back to all the HMIs. The HMI can communicate with each other with this driver. A protocol could be made where an HMI could instruct other HMIs to disable that feature temporarily and enable it again when the 'main' HMI is done controlling that system.

An example of mirroring can also be taken from the Efficiency truck, where there was an HMI on a tablet on the dashboard for the passenger, but also a monitor mounted at the top for the customer in the middle seat to have a look. The monitor could not be connected to the tablet, as the tablet was connected wirelessly. A special gateway received page messages from the tablet HMI and forwarded this command to the viewing HMIs to instruct it to show a specific page. This system added complexity and possible failures.

This could also be solved by having an echo driver. An example system is shown in Figure 4.10. This echo driver 'receives' any sent message, and the rest of the Software Gateway broadcasts it to all the connected HMIs. This makes adding a mirroring system easy and more stable, as it only requires sending or receiving a message to a 'bus' on an often already-running Software Gateway.

## 4.4 Mapping software to hardware

There are multiple methods to map the software components to the hardware components, some better than others. Decoding the messages before hardware

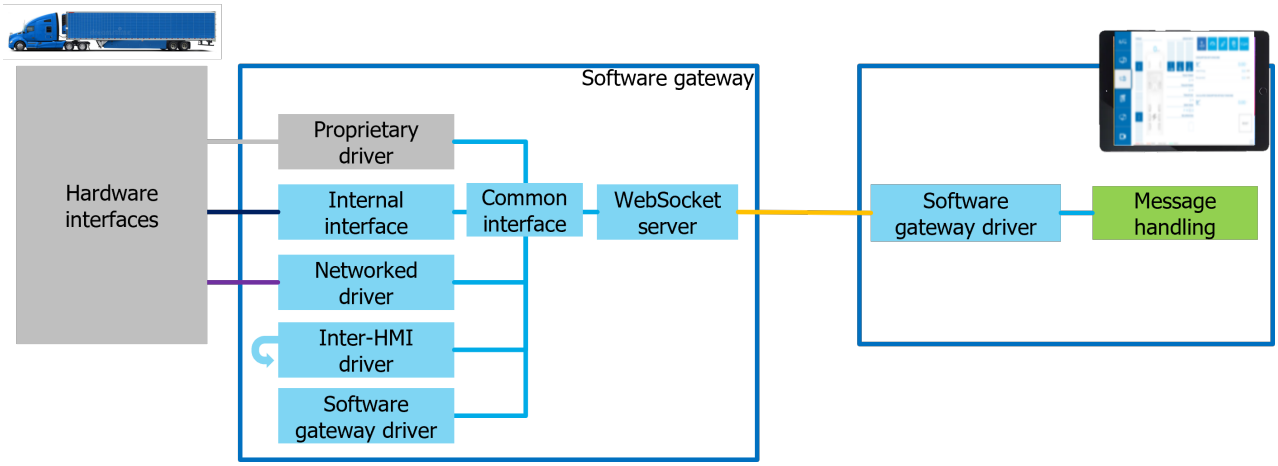



Figure 4.11: **Software gateway mapping**

interface drivers would be an illogical mapping. The criteria for the final mappings were stability, speed/efficiency, hardware interface usability, and reusability. An unstable system is of no use and would not be used. A decent speed and efficiency are also critical to not impact the HMIs too much and to not interfere with other networked systems. If no hardware interfaces can be used or it is only usable for a single system, then it is too limiting, like the current system.

These requirements resulted in the following two mappings, one with a Software Gateway and the other with a simple WebSockify gateway. These two mappings are compatible with each other, making it a very flexible architecture.

### 4.4.1 Software Gateway

The main mapping is by building a Software Gateway. This mapping is shown in Figure 4.11. This gateway would be an application that connects on one side to one or more of the hardware interfaces in the vehicle and on the other side HMIs connect to it. This gateway would have multiple drivers for the networked interfaces, internal interfaces, like SocketCAN, and the proprietary drivers included and thus can connect to all the possible hardware interfaces. To make it work for all the hardware interfaces, it would have to work on Linux, for SocketCAN, and on Windows, for some proprietary drivers, like the Vector XL driver.

The downsampling and cyclic broadcasting components are not shown in the architecture, but the common interface will have those features to be used by all the drivers.

The HMI would then connect to the Software Gateway through a WebSocket connection. To make configurations easier and centralised, the HMI would upload the configuration it wants with the used buses. The gateway would then connect to all the required buses and broadcast all the data to all the connected HMIs. The data can be broadcasted to all the HMIs, as they all have the same webpage running and all need the same information. This makes it simpler to build a Software Gateway, as it only needs to connect all HMIs to one set of

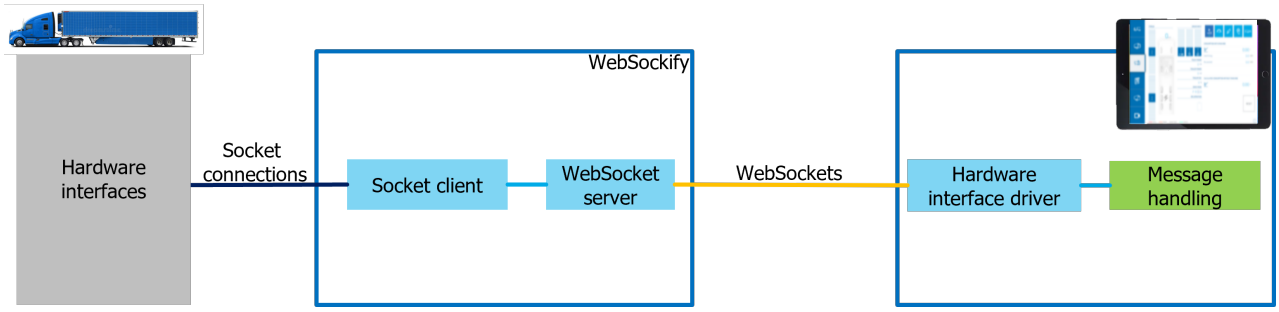



Figure 4.12: **Websocket gateway mapping**

buses and not keep track of what data of which buses are required by which WebSocket connection.

The HMI would need one driver for the gateway. Besides this driver, only a message parser is needed on the HMI. This makes it possible to only send the binary data over the WebSocket connections and not the bigger parsed data, resulting in lower network traffic and less computing needed on the Software Gateway.

Using a single connection to the hardware interfaces with the Software Gateway lowers the network load and processing load on the interfaces. The devices will also not receive duplicate or conflicting instructions from multiple connections, like when the HMIs would connect to it directly.

A custom application for streaming the data from the hardware interfaces to the HMIs also creates the possibility to add extra features like filtering and rate limiting to only forward the messages that the HMIs need for the visualisations.

When there are only networked interfaces, then the Software Gateway could even run inside a container, making it possible to run on a router or be easily deployed on any computer available in the vehicle.

It could be possible to build a Software Gateway for the tablets the HMIs are running on, but this is not recommended, as Android and iOS throttle apps that are running in the background, resulting in lower performance. The tablets can only connect to the networked hardware interfaces and are connected via Wi-Fi, limiting bandwidth and increasing latency.

### 4.4.2 WebSockify

Another mapping is by putting the hardware interface drivers on the HMI and having a WebSocket to Socket gateway between the HMIs and the networked hardware interfaces. This gateway can be created with WebSockify and can run on any computer and router in the network. This mapping would put the HMI in direct control of the hardware without a custom application in the middle, like in the mapping shown in Subsection 4.4.1.

Figure 4.12 shows the system of this mapping. The message handling, like decoding, is still done on the HMI, as there is no other part in the system capable of decoding. This mapping with multiple HMIs will result in multiple WebSocket connections, each having a Socket connection, resulting in multiple connections to each hardware interface. Multiple connections might result in conflicting

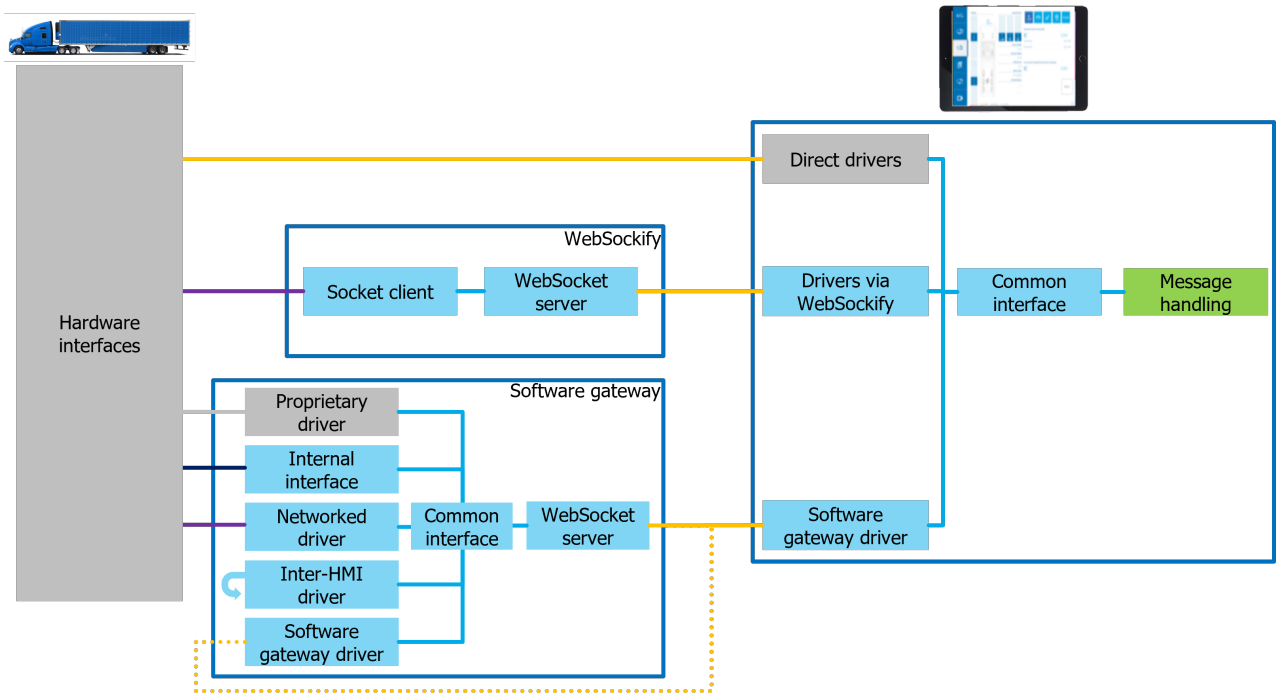



Figure 4.13: **Final global architecture**

instructions for the hardware. The current system has the same setup, where multiple HMIs send settings to a single device, which works without big issues.

For simple HMI systems with a small set of connected clients and not too much bus traffic, removing the need for filtering, this system would be great.

WebSockify can run in a container and thus also on a router, removing any requirement on extra components in the vehicles. This system is more flexible in selecting which HMI connect to which gateways, as it can connect to a hardware interface or not. This flexibility makes it possible to have one HMI connect to a gateway and another HMI not connect to it.

## 4.5 Final architecture

The proposed mappings from Section 4.4 are merged into a single final mapping. This final architecture is shown in Figure 4.13. Then the HMI JS code can connect, based on the configuration, to the simple WebSockify gateway and have the hardware driver on the HMI. It can also connect to a Software Gateway when there are buses that cannot be reached with WebSockify, when the downsampling systems are wanted, or when there are other features required from the Software Gateway.

There are also hardware interfaces with included WebSocket servers, like the *ESD* EtherCAN/2. These can be reached directly from the HMI, resulting in three methods for the HMI to communicate with hardware interfaces.

For the HMI, this would require it to have a similar system as the Software Gateway, where a common interface must be defined, and all the gateway drivers

must adhere to it. JS can have interfaces and inheritance, and such a common interface can make adding a new hardware interface easier. Using these common interfaces makes it easy to change the hardware setup without changing any code, only the configuration.

The communication to and from the hardware interfaces will always be using their protocol when using direct drivers and using WebSockify. The communication to the Software Gateway can be any protocol but should be optimised for the bus messages to optimise performance and keep the network load as low as possible. The proposed protocol has binary WebSocket messages, with flexible data length for the messages and puts multiple bus messages into a single WebSocket message. This way, the overhead of a single WebSocket message is spread over multiple bus messages. For configuration and status updates, the text channel of the WebSocket connection can be used with JSON messages. These JSON messages require more bandwidth than possibly needed, but they are more flexible and easier to generate and parse.

# Chapter 5

# Minimum Viable Product

To check that the proposed architecture from Chapter 4 works and to build a usable system for ZF, an MVP was built. This MVP is targeted to replace the current system, and thus a complete system must be made, including Software Gateway, WebSockify, and HMI code.

Firstly, the architecture (Section 5.1) of the MVP is shown and explained. Following that section, the used software tools and languages (Section 5.2) and software components (Section 5.3) are described. They contain information on the software used to build the system and some specific components. The gateway types are highlighted after that in Section 5.4. In Section 5.5, tests are shown to check that the MVP meets the requirements. Finally, Section 5.6 ends this chapter with conclusions and shows which requirements (Section 1.1) are met by the MVP.

The implemented hardware interfaces are listed in Appendix A with their drivers and how they could be used with the HMI system.

## 5.1 Architecture

The architecture of the MVP is the architecture as proposed in Chapter 4 but made specifically for the hardware available and requirements of the MVP. Figure 5.1 shows the global architecture. The hardware interfaces can only interact with the Software Gateway or WebSockify, except for the EtherCAN/2. This interface can also be used directly without any gateway.

The Software Gateway architecture, as shown in Figure 5.2a, has multiple drivers, conforming to a common interface. This interface makes it possible to easily add a new feature or add a new driver. Each bus has a driver and possibly a filtering and rate-limiting system, depicted with the downsampling systems box. Also, a broadcasting system for cyclically sending a message can be used with a bus. The messages from all the buses are packed into a single queue for the WebSocket system that broadcasts the data to all the connected HMIs.

The HMI architecture looks almost the same, as shown in Figure 5.2b, except that it does not have the downsampling and broadcasting systems. The yellow parts are components that the HMI developer should implement, as these are specific for each vehicle. The features that are not present in the HMI architecture compared to the Software Gateway architecture are not deemed useful and

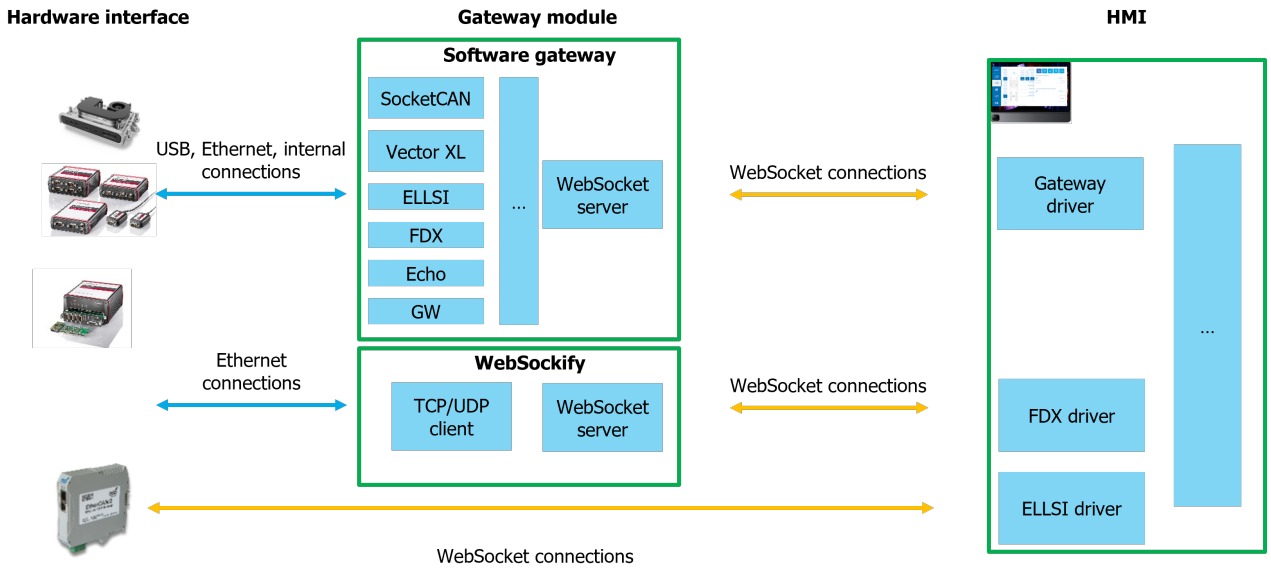


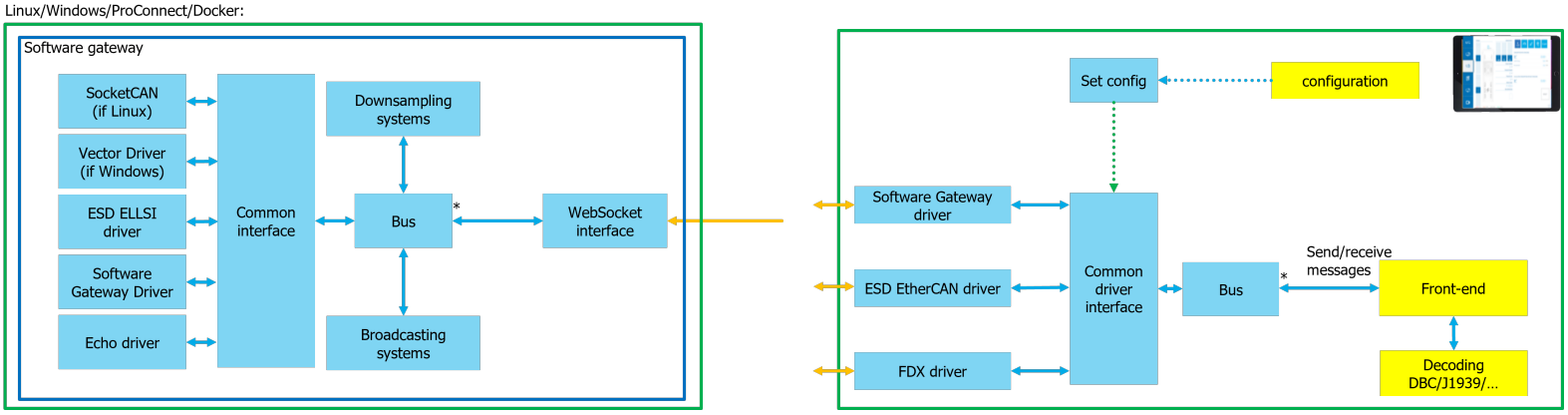

Figure 5.1: **Global architecture of the HMI system**

necessary. Filtering can be done by the decoding system and would not take much processing power away when it is implemented. Rate limiting on the HMI will not reduce the load on the HMI hardware. The HMI would have already received the messages by then and thus not reduce CPU or network load. Cyclic broadcasting to a vehicle bus from the HMI should be done by the front-end code, as the developer wants to implement it there instead of adhering to the different requirements of the HMI system. The other option for cyclic broadcasting would be to use a Software Gateway. Decoding the messages depends on the DBC, and some HMI projects need extra decoding features, like putting multiple 8-byte messages together into a single larger message.

## 5.2 Software tools

For the MVP, two systems, the HMI code and the Software Gateway, that cooperate had to be made. The JS code on the HMI has to work with WebSockify and the Software Gateway to receive and send messages from all the buses. The JS software also has to include the hardware drivers for the interfaces that are reachable through WebSockify. The Software Gateway has to run on multiple hardware targets and operating systems selected for the MVP, connect to the networked hardware interfaces, and include the possible proprietary drivers on some platforms. The selected hardware for the Software Gateway is Windows and Linux x86 computers, Docker x86, ARMv7, and ARMv8 containers, the ZF ProConnect (Linux ARMv8) and Nvidia Jetson Nano (ARMv8). These parts were available and interesting to use, and some are already running in innovation vehicles.

For the HMI JS library, plain JS was required as the programming language, as this is the only language possible for websites. No frameworks or libraries




(a) **Architecture of the Software Gateway** (b) **HMI JS architecture**

Figure 5.2: **Different parts of the architecture for the HMI system**

were added to the HMI code to make it not constrained to that single framework and to keep it lightweight. No libraries makes it also easier to include the JS code into an HMI project. The separate parts of the HMI code are organised in separate ECMAScript (ES) modules, as this is the official way to package JS components. These ES modules and thus also the HMI gateway code, can run in Node.js, making it possible to build an application running outside a browser to interface with the HMI system. For the tests of the MVP, these modules were also used with Node.js. Running the tests outside a browser eliminates problems browsers might give by throttling webpages when they are not in the foreground, for example.

CanViewer2, a special HMI to see current bus data for debugging and a proof of concept of this project, uses Vue.js as the framework for visualisation, as this is an easy framework for building web apps. It is a reactive framework, where a variable change is immediately updated in the graphical user interface (GUI), and it uses templates, which is great when having duplicate parts, like showing a list of bus messages. Another great thing about Vue is the build functionality. This build functionality takes all the JS files and minifies and removes all dead code (tree-shaking), resulting in a small webpage consisting of only a few files. It was decided that CanViewer2 could be implemented with a framework, as this is a standalone webpage and will not be included in an HMI for the customers to see.

For the Software Gateway, a programming language with support for multiple architectures and operating systems had to be chosen. As there were also some drivers provided in C and C++ and these languages also compile to machine code and are not interpreted, like JS, C++ was chosen. This is a programming language with support for object-oriented programming with special resource management features. This makes working with resources like memory, sockets and objects easier than doing that in C. C++ does not have WebSocket components, like JS has, natively, but those features can be imported by using libraries. The Software Gateway is allowed to have libraries. Otherwise, it would require more development time. The Software Gateway is not imported into any other system, like the JS code. It is possible to use those libraries by just adding

the files to the source code and including them, but some libraries that were used, like Boost, are not straight-forward to compile and need extra setup. To make it easier to compile for the multiple targets and build pre-compiled packages, Conan was selected as the package manager. Conan has a repository of pre-compiled libraries, making it possible to add a package easily and have it compiled without extra effort.

The used packages were Boost, for argument parsing, thread pools and filesystem operations, WebSocket++ for WebSocket server and client features. Finally, Niels Lohmann's JSON library was included for JSON parsing and serialisation.

CMake was chosen as the build-system for C++ to make it easier to select the correct compiler. It needs one configuration file for all the different targets and combines all the drivers and Conan packages into a single executable or Windows installer.

## 5.3 Software components

The architecture of Figure 5.2 is used to develop the MVP. Some of these components are highlighted in this section. The bus messages on the WebSocket connections are implemented as proposed in Figure D.1.

### Broadcasting

Broadcasting was implemented as envisioned in Section 4.1.4. The configuration that the HMI has can include a broadcast configuration to set default data, a sending interval and, optionally, a timeout. The timeout should be used when the message is for a system that should revert to a default state when an HMI is disconnected.

### Software Gateway driver

The Software Gateway driver in the Software Gateway works like any other bus driver. This driver can connect to another Software Gateway and acts as an HMI to receive and send messages. It was chosen to implement this as a normal driver to keep a clean architecture and system. The driver opens a WebSocket connection to the other Software Gateway like any HMI. It was chosen to use WebSockets for this communication, even though it could be done with the lower-level TCP. However, this would require the Software Gateway to also have a TCP interface besides the WebSocket interface. Acting as a normal HMI makes it simpler, and the WebSocket overhead is not much. It would also be more firewall-friendly, as it looks like regular website traffic.

### Downsampling

The MVP also includes downsampling with filters and rate limiting. The implemented filters are lists of IDs and ranges. Rate limiting is done with the time slice algorithm shown in Figure C.1b. It stores a list of messages that are sent in the last time slice. When a message is received, this list is checked, and when it is not yet forwarded, it is forwarded. Otherwise, it is discarded. At every interval, this list is reset.

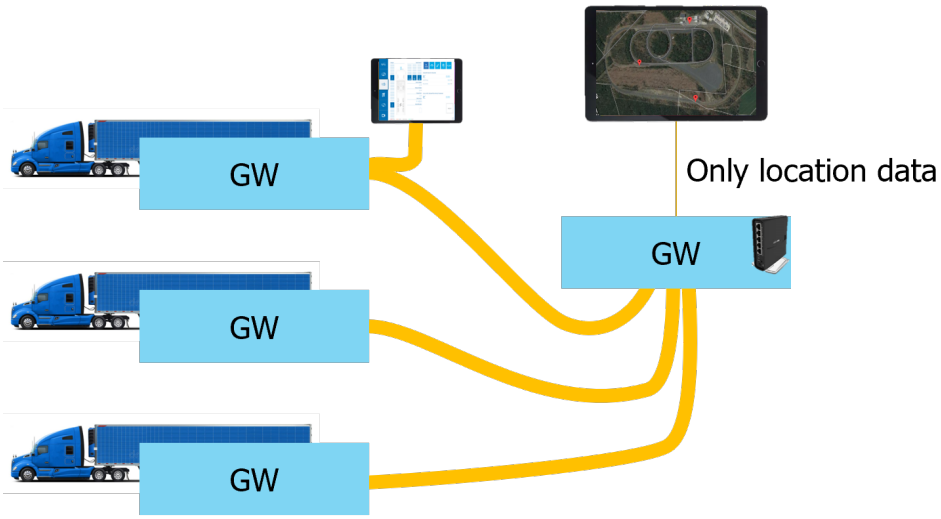



Figure 5.3: **Software Gateway filtering illustration. A thicker line means more data**

Each bus has its own filter and rate-limiting system. This makes it possible to have different rates and different filters. The rate is the same for all messages from a bus as the algorithm is time slice triggered.

Even the Software Gateway can have filtering or rate limiting enabled. This could be useful for systems with HMIs inside the vehicle and an HMI showing the locations of the vehicles, as illustrated in Figure 5.3. Then a Software Gateway (map gateway) could be placed between the map HMI, connecting to the Software Gateways (vehicle gateways) inside the vehicles. The map HMI would only need the location, so the map gateway could filter out the other messages. This cannot be done on the vehicle gateways because the messages for the HMIs inside the vehicles would also be filtered, as the filtering is done before the broadcasting to all the connected HMIs.

## 5.4 Software gateway hardware

When using the new HMI system, the Software Gateway is the application that everything connects to. Only in smaller HMI systems with specific hardware, the Software Gateway is not required, and WebSockify could be used. In those smaller systems the Software Gateway could still improve stability and reduce network and CPU load by controlling the hardware interfaces from one single spot. Therefore the Software Gateway needs to be able to run on as much hardware as possible to reduce the costs and space of adding computers just for this purpose to the vehicles.

The Software Gateway was developed in C++ without any dependencies on the operating system besides the standard library. The only library that it includes, when available, is the SocketCAN header. The other dependencies, like Boost and the JSON libraries, do not have any extra requirements. This makes it possible to compile easily for any Linux distribution and also makes it

possible to compile for Docker.

For Windows, there are three possibilities to get the Software Gateway up and running. The first is running the Software Gateway in WSL. This is a kind of virtual machine, but more integrated into Windows and was used for development. Using WSL, it makes it possible to run Linux applications on Windows with a smaller overhead than a real virtual machine. For network-connected hardware interfaces, this will work as fine as running it in Docker, as it does not require other Operating System (OS) or hardware capabilities. When it is required to use SocketCAN devices connected through USB on Windows, then a kernel supporting SocketCAN and USB-passthrough will make it possible. The second option is using the Software Gateway compiled for Windows. This will enable the Vector XL driver but disable the SocketCAN driver. To make it easy to start on Windows, an installer installs the gateway, webserver and CanViewer2. This way, it is not required to install a complete toolbox.

Docker is the third option for Windows (using WSL) and also an option for Linux OSs and for the routers often used in the vehicles, the MicroTik HapAC$^2$. Using Docker, it is possible to compile for other architectures, like ARMv7 and ARMv8, without the hassle of cross-compiling. This is done by emulation at image build time. This emulation makes compiling a bit slower. The Dockerfile, the instructions for building an image, is split into two parts. The first stage compiles the Software Gateway using all the required build tools, and then in the second stage, the application is copied to a fresh Alpine image, one of the smallest possible. This reduces the size from 480-750 MB to 9-14 MB, depending on the target architecture. The downside of using Docker is that it only supports networked hardware interfaces, but it allows one to easily set up a lightweight Software Gateway.

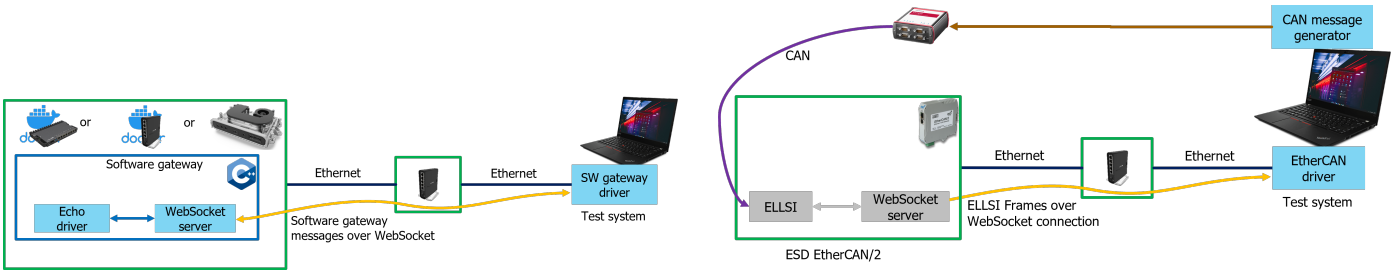
Using cross-compilers for specific targets, like the ZF ProConnect, it is possible to compile the Software Gateway on a normal computer and transfer the applications to the target hardware that does not have the capabilities to compile. Conan and CMake support this by making it easy to switch targets, setting special flags and detecting the compiler and target.

## 5.5 Tests

To test the system, the following tests were done to compare the MVP to the current solution as described in Chapter 3 and to check the requirements as set in Section 1.1.

### Boot time

The boot times of the testable hardware components are shown in Table 5.1. The target boot time is 30 seconds. This table shows that the ProConnect and EtherCAN/2 are the fastest, and the two tested routers are way slower. The routers are probably slower because they have less computing power, and the Software Gateways are run in a container system, also required to start. Starting a container on the routers takes only a few seconds. The boot time is tested in the setup as shown in Figure 5.4a and Figure 5.4b. Booting is considered done when the first message is received at the test system.




(a) **Test setup for the Software Gateway** (b) **Test setup for the *ESD* EtherCAN/2**

Figure 5.4: **Test setups to test the boot-time, throughput, and overhead**

| Hardware | ZF ProConnect | MicroTik HapAC$^2$ | MicroTik RB5009 | *ESD* EtherCAN/2 |
|---|---|---|---|---|
| Boot-time | 14s | 84s | 66s | 22s |
| Message throughput (echo, 64 byte messages) | 159,000/s | 45,000/s before crashing | 159,000/s | - |

Table 5.1: **Boot-time and throughput of some Software Gateway hosts**

Normal Linux and Windows computers can have a boot time less than 30 seconds, but that depends on the hardware. The boot-time of the Software Gateway is less than a second. Each added bus in the configuration adds up to 1 second to the boot-time, depending on the type.

## Throughput and overhead

Table 5.1 also shows the throughput when using an echo bus with a single HMI that is continuously sending 64 bytes messages. The ProConnect and MicroTik RB5009 router have the same throughput, as it is at the limit of the network, around 100Mbps. CAN buses can transfer up to around 3500 8 byte messages per second when using 500k baud, so this throughput is enough for several buses. The MicroTik HapAC$^2$ rebooted when it had too much traffic, but further tests show that it can forward 45,000 messages per second without crashing. These tests are also done with the setup of Figure 5.4a.

Throughput alone is not a good measure of efficiency, but network efficiency is also important to not interfere too much with other network streams. These additional streams include video streams that take a lot of network bandwidth and are sensitive to latency. Many messages are sent for 20 seconds and echoed back to an HMI to test network efficiency. The received network usage is then divided by the number of messages. Only the received network traffic is measured, as more messages will be received than sent in a normal HMI system, and the HMI code only combines messages in a single WebSocket message when they are sent in the same library call. The overhead is the number of bytes more than required. The required bytes include four for the message ID and a specific number of data bytes. Rate-limiting was disabled for these tests, as the tests focused on throughput and overhead. Enabling rate-limiting would reduce network and HMI CPU load.

The *ESD* EtherCAN/2 was tested by receiving CAN traffic from a real bus, as shown in Figure 5.4b. Testing throughput would not test the capabilities of

| Gateway | Software Gateway | | | *ESD* EtherCAN/2 | |
|---|---|---|---|---|---|
| Message size (+ 4 bytes message ID) | 0 | 8 | 64 | 0 | 8 |
| Messages received | 11,947,600 | 8,333,900 | 3,302,600 | 20,954 | 19,960 |
| Received traffic (KB) | 87,499 | 136,200 | 239,294 | 748 | 733 |
| Traffic per message (bytes) | 7.3 | 16.3 | 72.5 | 35.7 | 36.7 |
| Overhead (bytes/message) | 3.3 | 4.3 | 4.5 | 31.7 | 24.7 |

Table 5.2: **Overhead of Software Gateway protocol compared to the *ESD* EtherCAN/2**

the hardware, but is capped at the CAN message upper limit.

The results of these tests are shown in Table 5.2. The overhead is between 3.3 and 4.5 bytes per message. This overhead includes one byte for data length, one byte for bus number, and one byte for flags per message. Besides overhead per message, also the configuration uploading, status sending and WebSocket overhead, this comes out to 0.3 to 1.5 bytes per message. The overhead is higher with larger messages, as less CAN message fit inside a WebSocket message to share the WebSocket overhead.

Table 5.2 also shows that the protocol that is used by the current system, the *ESD* EtherCAN/2, does not use message minimising and has messages that are always the same size. This protocol also has more bytes overhead, even when sending the longest CAN message possible (8 bytes). This shows that the new Software Gateway is more efficient than the current system and that the Software Gateway should be used when there is more than one HMI to keep the network traffic the lowest.

### Conclusion

These tests show that some hardware is fast enough to boot itself and the Software Gateway within the required 30 seconds. The routers are slower, but this is not a problem because the routers are also necessary for the network to work. Without the network, the HMIs cannot connect to hardware interfaces and must wait on the routers to boot.

The ZF ProConnect and MicroTik RB5009 can transfer more than 150000 messages per second but are limited by the network speed. This message rate is more than 40 CAN buses can generate. The overhead of the Software Gateway message protocol is much lower than the overhead of the *ESD* EtherCAN/2 protocol.

## 5.6 Conclusions

A Minimum Viable Product (MVP) was built according to the envisioned architecture from Chapter 4. The realised architecture is shown in Figure 5.5.

As shown in previous sections, the MVP can connect to multiple buses and be used for more complex systems, like inter-HMI communication, combining data from multiple vehicles and remote viewing without additional complexity.

The MVP can connect to four hardware interface types. First, it works with the Vector XL driver interfaces, like the VN7640. These devices and SocketCAN

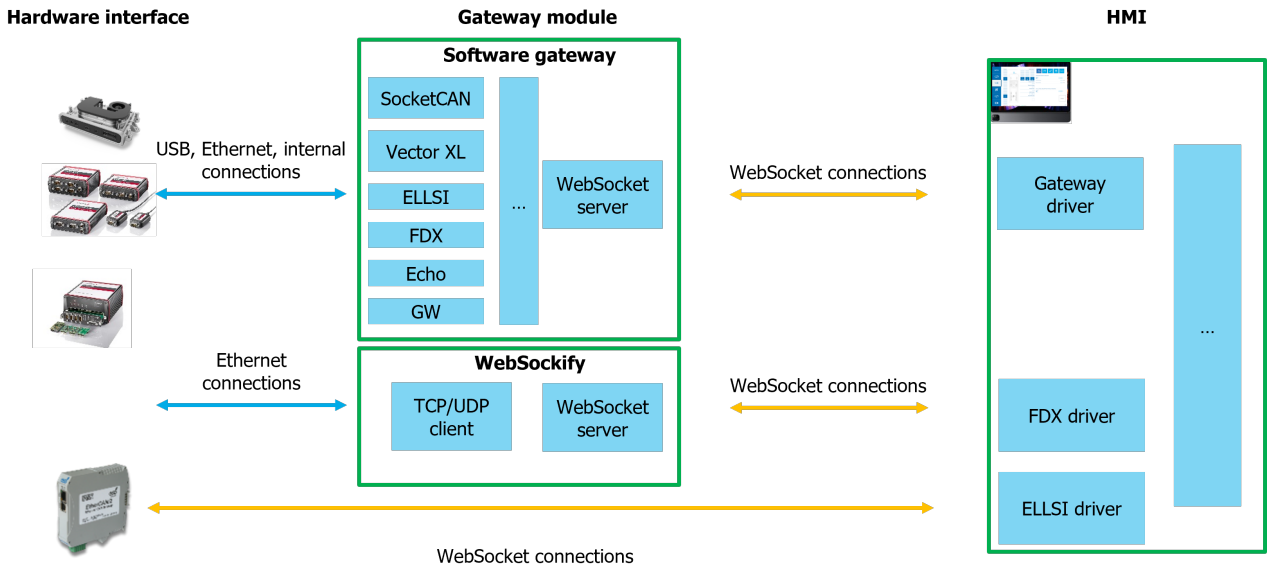



Figure 5.5: **MVP architecture**

interfaces, like the ZF ProConnect, are only usable when using the Software Gateway. The third type is interfaced with the Vector FDX protocol, like the VN8914. The last hardware interface is the *ESD* EtherCAN/2. The MVP implements a system for on the Human Machine Interface (HMI) as well. This system can directly connect to the Software Gateway and EtherCAN/2 devices. It can also communicate with the Vector FDX hardware interfaces using WebSockify.

The Software Gateway supports cyclic sending (broadcasting), filtering, rate limiting, and multiple buses simultaneously. It can also receive any sent message using the echo driver, supporting a basic inter-HMI communication for coordination or HMI page mirroring.

The code for the HMI and Software Gateway is made to enable easily adding a new driver for new hardware interfaces, removing the current need for buying a hardware interface when there is one already in a vehicle.

Tests show there is not much overhead per message and that message packing and minimising improves the throughput and efficiency. The tests also show that both systems are capable of passing more messages than any bus can generate.

## Requirements

Figure 5.6 shows the requirements, as set in Section 1.1, to indicate which are met and which are not yet included in the MVP. The items in green are the requirements included with the MVP. The requirements in red are not in the current MVP, mainly because they are not closely related to the core HMI system and are supportive features. The items still in black are features that are met depending on the system used, as explained below.

It was impossible to test FlexRay and Automotive Ethernet, as no hardware interfaces were available with those buses. FlexRay can be used with the FDX

hardware interfaces, but this is not tested.

The configurability section (item 3) was not completely fulfilled, as network settings setups are different for Windows, Linux and Docker. Often containers cannot change network settings, as the host system sets the IP. A Software Gateway running in Windows Subsystem for Linux (WSL) can also not change the host Windows's network settings. These issues make it challenging to make one system capable of controlling network settings on different hosts.

A website for configuration is not needed, besides the network configurations, as the HMI will send the configuration with the required buses. Therefore there is no need to have a webpage to set this up. A debug site is developed that shows received and decoded messages and makes it possible to send messages as well. This CanViewer2 also indicates the state of the connected gateways and interfaces.

The HTTP server from requirement item 2b is implemented as a separate application to improve stability and keep both applications specific to the domains they are made for.

Some of the hardware interfaces meet the hardware requirement (item 4). The ZF ProConnect boots in a few seconds and meets the size requirements. The MicroTik HapAC$^2$ does meet the size and housing requirements but takes longer to boot. These two devices also meet the supply voltage requirement. When using the Software Gateway on a Windows or Linux computer, it will probably meet the boot-time requirement but might not meet the size or supply voltage requirement.

The licensing requirement (item 5c) is met by all (currently) possible configurations, except when using the *Vector* VN8914. This hardware interface requires setup once in a dedicated application to set up the message definitions. When this interface is already included in the vehicle, this application is also already in use, so no additional licenses are required to use it for an HMI.

Meeting the per-project budget requirement (item 5b) depends on which buses are required for the HMI and what hardware is already included. The ZF ProConnect does not meet the small project requirement, but if it is already mounted in the vehicle for something else, it is cheaper to reuse it for the HMI than to add new parts.

Table 5.3 shows a decision matrix with the four different hardware interfaces that can be used with the MVP. It shows what is required, which buses are available and the reusability of the components. As each system is possible with the Software Gateway, the reusability and pros/cons do not consider that the router can be used for more things and that all systems are capable of filtering, rate limiting and cyclic broadcasting.

1. Vehicle Connectivity:
   (a) CAN bus
      i. Normal & CAN-FD speeds
      ii. 11-bit & 29-bit message IDs
      iii. Configurable speeds
   (b) LIN
   (c) Preferred: FlexRay & Automotive Ethernet
2. HMI Connectivity:
   (a) WebSocket server:
      i. Bi-directional and multiple HMI connections
      ii. Message decoding module in JavaScript (JS)
      iii. Filtering on message/bus, whitelisting
      iv. Efficient messaging, bundling of messages
      v. downsampling of messages
      vi. <75ms (13Hz) roundtrip time
   (b) HTTP-server:
      i. Serve HMI and debug pages
      ii. >500MB size possible
3. Configurability:
   (a) Modular system
   (b) Centralised
   (c) Network settings
   (d) Website for configuration
   (e) HTTP-server settings
4. Hardware:
   (a) Production type hardware
   (b) boot-time: <30s
   (c) Must have a housing
   (d) Size: 20x30x10cm to fit easily in vehicles
   (e) Supply voltage 10-32V
5. Other:
   (a) Website with state of different modules
   (b) Per-project budget
      i. Small HMI project: <€1k
      ii. Big HMI project: <€5k
   (c) No software licensing per developer or device

Figure 5.6: **Requirements as set in Section 1.1 with green items that are included in the MVP and red items that are not included.**

| Type: | ***ESD* EtherCAN/2** | **Vector FDX** | **SocketCAN (ZF ProConnect)** | ***Vector* VectorXL** |
|---|---|---|---|---|
| No of boxes: | 2: EtherCAN/2, router | 2: VN8914, router | 1: ProConnect | 3: VN7640, Windows PC, router |
| Hardware price: | €600: 500 + 100 | €8100: 8000 + 100 | €1800 | €1800: 1200 + 500 + 100 |
| Ease of use: | Easy | Needs FDX config setup | Easy | Medium, needs driver setup |
| Reusability | None | None | Device can be used for other tasks | PC can be used for other tasks |
| Buses | 1x CAN | 3x CAN, LIN, FlexRay, Automotive Ethernet | 2x CAN, 2x CAN FD, Automotive Ethernet | 3x CAN (FD), FlexRay |
| Pro | Existing, simple and proven system | More buses | Lots of buses<br>Dedicated hardware<br>Can act as access point | Some buses |
| Con | Just a single bus | Requires setup for each message | SocketCAN requires Linux | Windows required for the Vector XL driver |

Table 5.3: **Decision matrix for the implemented hardware**

# Chapter 6

# Conclusions

As stated in Chapter 1, the project goal was to determine how to (re)design a Human Machine Interface (HMI) system for future innovation vehicles. The first step was analysing the preferred automotive buses, namely Controller Area Network (CAN), Controller Area Network with Flexible Data-rate (CAN FD) and other future networks. These automotive networks are all message-oriented instead of service/connection-oriented, like standard networking. This message-oriented approach requires the HMI system to handle simple messages instead of more complex connections/services. Another requirement is that it should be reasonably efficient in terms of load on the internet backbone, as that is shared with other systems, and the HMIs should not need processing power.

A range of hardware interfaces was researched on communicating and making a complete system with those components. The software components were also investigated on what is needed and how to make it work for a flexible and performant system. Also, other use cases like distributed systems and inter-HMI communication were investigated. The next step was investigating the other hardware and its capabilities, like virtualisation. Two mappings were made to map the software components onto the hardware components, keeping efficiency and flexibility in mind. These two mappings were combined into a single architecture to join the possibilities and flexibility. This final architecture, as shown Figure 6.1, is proposed to have a flexible and reusable system for receiving and sending messages on different buses. The HMI can connect directly to hardware interfaces if they support WebSocket connections. When a hardware interface has a Socket interface, then WebSockify can convert WebSocket messages for the HMI to the TCP/UDP messages the hardware interfaces use. The Software Gateway can connect to any hardware interface, networked or otherwise. The Software Gateway can provide extra features, like filtering, rate limiting, and cyclic broadcasting.

The proposed architecture allows any capable hardware to be easily integrated into the system and keeps the complete system simple.

To test the architecture, a Minimum Viable Product (MVP) was made as described in Chapter 5. The MVP includes drivers for four hardware interfaces and a software gateway to interact with these interfaces. This software gateway can be run on Windows, Linux and any other hardware capable of running Docker and is thus flexible on where to be located. The MVP connects to four hardware interfaces, namely the *ESD* EtherCAN/2, *Vector* VN7640, *Vector* VN8914 and

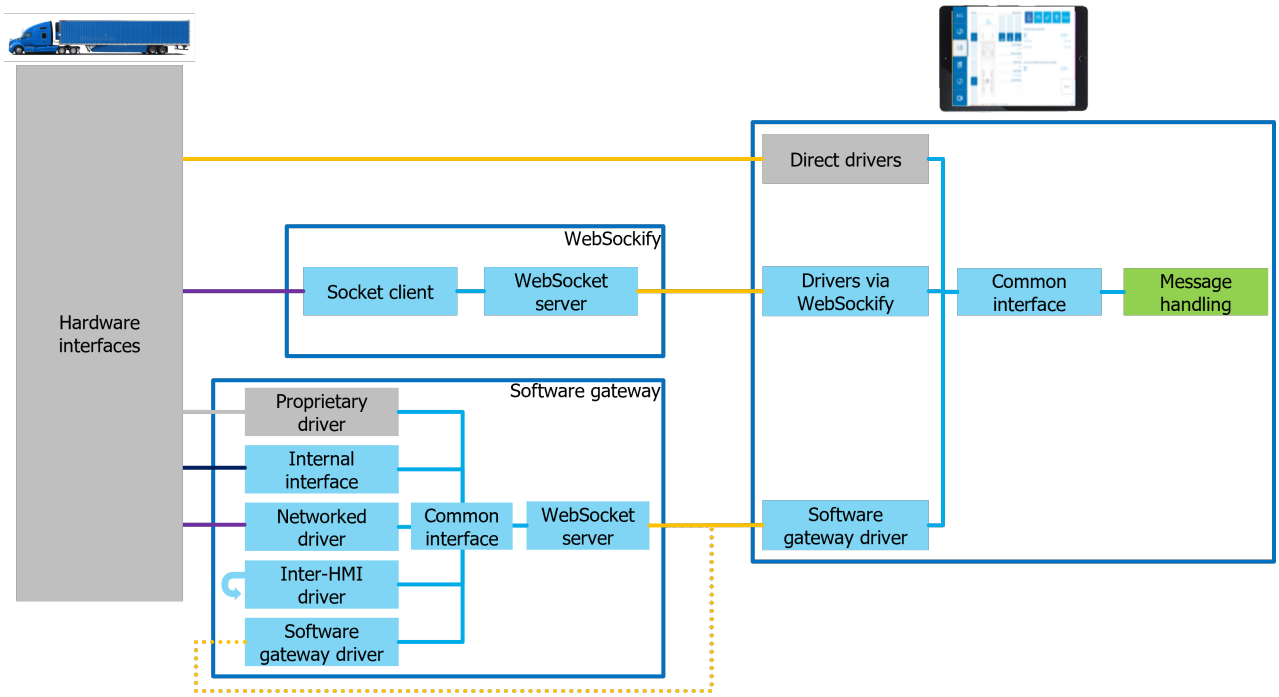



Figure 6.1: **Global architecture**

SocketCAN devices, like the ZF ProConnect. With this hardware, the MVP can connect to CAN and CAN FD, as required for this project, and is also able to connect to FlexRay, Local Interconnect Network (LIN) and other buses with some external configuration. The MVP meets most of the requirements set by ZF. The tests show that the networking overhead of the Software Gateway to HMI is lower than the current system.

Therefore, the chosen architecture is acceptable for this problem as the MVP shows that it works and does not add overhead over the current system while adding various possible hardware interfaces.

The MVP allows developers to interact with more vehicle buses and more types of buses and make distributed systems, using inter-HMI communication and by combining data from multiple vehicles.

# Chapter 7

# Future Work

There are some leftover tasks to complete the system from the current MVP. These tasks include testing the different components and analysing the code for possible faults, like deadlocks and exceptions, to ensure a stable system.

Besides making the system more stable, more drivers for more hardware interfaces can be added, like the *Ixxat* or *Peak-System* compatible product lines. Also, support for even more networks could be added. One interesting future network is Controller Area Network Extra Long (CAN XL), which has even longer messages, a different addressing system and indicators for higher-layer protocols. These features are possible with the designed architecture.

Another network that could be added to the MVP is Automotive Ethernet. As this is a service-oriented network without a common bus, snooping and grabbing all the data is impossible. Therefore a different approach must be taken to get data from and to the vehicle. When data is received in the Software Gateway, it can be processed like any other message.

Another feature that was requested but did not make it entirely to the MVP is inter-HMI communication for coordination and control. This feature could ensure that only one HMI is sending a message or has control of a feature without interference from the other HMIs. The Software Gateway has a basic feature for this in the form of the *echo* driver, which returns all incoming messages to all the connected HMIs, but a distributed protocol on all the HMIs is needed to have a robust control.

This project was built around the fact that hardware was available and software was required to connect it to the HMI. A possible future project for ZF could be building hardware based on the requirements instead of conforming the software to the available hardware. A hardware interface could, for example, be made with an Espressif ESP32, as it has CAN, Wi-Fi/Ethernet and WebSocket support and a CAN FD controller with SPI, like the Microchip MCP251863, is easily added.

# Bibliography


[1] Charles Anderson. Docker [software engineering]. *IEEE Software*, 32(3):102–c3, May 2015.

[2] CAN in Automation. Can fd - the basic idea. `https://www.can-cia.org/can-knowledge/can/can-fd/`. Last accessed: Sep. 15, 2022.

[3] Steve Corrigan. Introduction to the controller area network (can). `https://www.ti.com/lit/an/sloa101b/sloa101b.pdf`, May 2016. Last accessed: Jan. 16, 2023.

[4] Irina Costachescu. 101: Local interconnect network (lin). `https://community.nxp.com/t5/Blogs/101-Local-Interconnect-Network-LIN/ba-p/1284877`, May 2021. Last accessed: Jan. 21, 2023.

[5] Docker Inc. About storage drivers. `https://docs.docker.com/storage/storagedriver/`, August 2021. Last accessed: Oct. 13, 2022.

[6] Martin Falch. Can dbc file explained - a simple intro. `https://www.csselectronics.com/pages/can-dbc-file-database-intro`, 2022. Last accessed: Oct. 12, 2022.

[7] Martin Falch. Can fd explained - a simple intro [2022] - css electronics. `https://www.csselectronics.com/pages/can-fd-flexible-data-rate-intro`, March 2022. Last accessed: Sep. 15, 2022.

[8] Martin Falch. J1939 explained - a simple intro [2022]. `https://www.csselectronics.com/pages/j1939-explained-simple-intro-tutorial`, 2022. Last accessed: Oct. 12, 2022.

[9] Chindris Gabriel and Horia Hedesiu. Integrating sensor devices in a lin bus network. In *26th International Spring Seminar on Electronics Technology: Integrated Management of Electronic Materials Production*, pages 150–153. IEEE, May 2003.

[10] Gupta. Docker "copy-on-write (cow)" strategy. `https://stackoverflow.com/a/71480874`, March 2022. Last accessed: Jan. 22, 2023.

[11] Florian Hartwich. Can with flexible data-rate. In *Proc. iCC*, pages 1–9. Citeseer, March 2012.

[12] Florian Hartwich. Introducing can xl into can networks. https://www.can-cia.org/fileadmin/resources/documents/proceedings/2020_hartwich.pdf, July 2020. Last accessed: Sep. 15, 2022.

[13] Zongmin Jiang, Yan Chang, and Xuefen Liu. Design of software-defined gateway for industrial interconnection. *Journal of Industrial Information Integration*, 18:100130, 2020.

[14] Rowena Jones and Gregoire De Turckheim. Flexray module training. https://blog.scaleway.com/iot-hub-what-use-case-for-websockets/, February 2021. Last accessed: Jan. 22, 2023.

[15] National Instruments Corp. Flexray automotive communication bus overview. https://www.ni.com/nl-nl/innovations/white-papers/06/flexray-automotive-communication-bus-overview.html, June 2021. Last accessed: Sep. 23, 2022.

[16] Victoria Pimentel and Bradford G Nickerson. Communicating and displaying real-time data with websocket. *IEEE Internet Computing*, 16(4):45–53, August 2012.

[17] Stefan Profanter, Ayhun Tekat, Kirill Dorofeev, Markus Rickert, and Alois Knoll. Opc ua versus ros, dds, and mqtt: performance evaluation of industry 4.0 protocols. In *2019 IEEE International Conference on Industrial Technology (ICIT)*, pages 955–962. IEEE, February 2019.

[18] Andrea Reindl, Daniel Wetzel, Norbert Balbierer, Meier Hans, Michael Niemetz, and Sangyoung Park. Comparative analysis of can, can fd and ethernet for networked control systems. In *Embedded World Conference 2021*, Nürnberg, Germany, October 2021.

[19] Robert Bosch GmbH. Can xl. https://www.bosch-semiconductors.com/ip-modules/can-protocols/can-xl/. Last accessed: Sep. 15, 2022.

[20] Robert Bosch GmbH. Can xl - the next step in can evolution. https://www.bosch-semiconductors.com/media/ip_modules/pdf_2/can_xl_1/20220825_can_xl_overview_v2.pdf, August 2022. Last accessed: Sep. 15, 2022.

[21] Rudy Tellert Elektronik. Bit position of can signals (start bit). https://www.tellert.de/files/media/temes/help/position-of-can-signals-(start.html, 2017. Last accessed: Oct. 12, 2022.

[22] Robert Shaw and Brendan Jackman. An introduction to flexray as an industrial network. In *2008 IEEE International Symposium on Industrial Electronics*, pages 1849–1854, Cambridge, UK, November 2008. IEEE.

[23] Benfano Soewito, Christian, Fergyanto E.Gunawan, Diana, and I Gede Putra Kusuma. Websocket to support real time smart home applications. *Procedia Computer Science*, 157:560–566, October 2019.

[24] Branislav Sredojev, Dragan Samardzija, and Dragan Posarac. Webrtc technology overview and signaling solution design and implementation. In *2015 38th international convention on information and communication technology, electronics and microelectronics (MIPRO)*, pages 1006–1009, Opatija, Croatia, May 2015. IEEE.

[25] Texas Instruments. Flexray module training. `https://www.ti.com/lit/ml/sprt718/sprt718.pdf`, November 2015. Last accessed: Jan. 21, 2023.

[26] UAB Elektromotus. Elektromotus can bus topology recommendations. `https://emusbms.com/files/bms/docs/Elektromotus_CAN_bus_recommendations_v0.2_rc3.pdf`. Last accessed: Sep. 15, 2022.

[27] Rufin VanRullen, Leila Reddy, and Christof Koch. The continuous wagon wheel illusion is associated with changes in electroencephalogram power at 13 hz. *The Journal of neuroscience : the official journal of the Society for Neuroscience*, 26:502–7, February 2006.

[28] Vector Informatik GmbH. Can: Bitstuffing. `https://elearning.vector.com/mod/page/view.php?id=51`, September 2021. Last accessed: Sep. 15, 2022.

[29] Peng Zhang. *Advanced industrial control technology*, chapter Industrial control networks. William Andrew, 2010.

# Acronyms

**API** application programming interface.

**BCM** Broadcast Manager.

**CAN** Controller Area Network.

**CAN FD** Controller Area Network with Flexible Data-rate.

**CAN XL** Controller Area Network Extra Long.

**CI/CD** Continuous Integration / Continuous Delivery.

**CRC** Cyclic Redundancy Check.

**CSMA** Carrier Sense Multiple Access.

**DBC** CAN database.

**ECU** electronic control unit.

**ES** ECMAScript.

**FDX** Fast Data Exchange.

**GUI** graphical user interface.

**HMI** Human Machine Interface.

**IoT** Internet of Things.

**JS** JavaScript.

**JSON** JavaScript Object Notation.

**kbps** kilobit per second.

**LIN** Local Interconnect Network.

**Mbps** megabit per second.

**ms** millisecond.

**MSB** Most Significant Bit.

**MVP** Minimum Viable Product.

**NUC** Next Unit of Computing.

**OPC UA** OPC Unified Architecture.

**OS** Operating System.

**PLC** Programmable Logic Controller.

**QoS** Quality of Service.

**ROS** Robot Operating System.

**SBC** single-board computer.

**SDT** service data unit type.

**TDMA** Time Division Multiple Access.

**UART** Universal Asynchronous Receiver Transmitter.

**VM** virtual machine.

**VPN** Virtual Private Network.

**WebRTC** Web Real-Time Communication.

**WSL** Windows Subsystem for Linux.

# Appendix A

# Hardware interfaces

The following hardware interfaces were chosen to be added to the MVP because they were available and most promising/interesting for future innovation vehicles.

- *ESD* EtherCAN/2
- *Vector* VN7640 (Vector XL driver)
- *Vector* VN8914 (Vector FDX)
- ZF ProConnect (SocketCAN)

In the following sections, the hardware interface and its provided driver are explained, how to connect to it and a corresponding system setup is shown that works with the MVP.

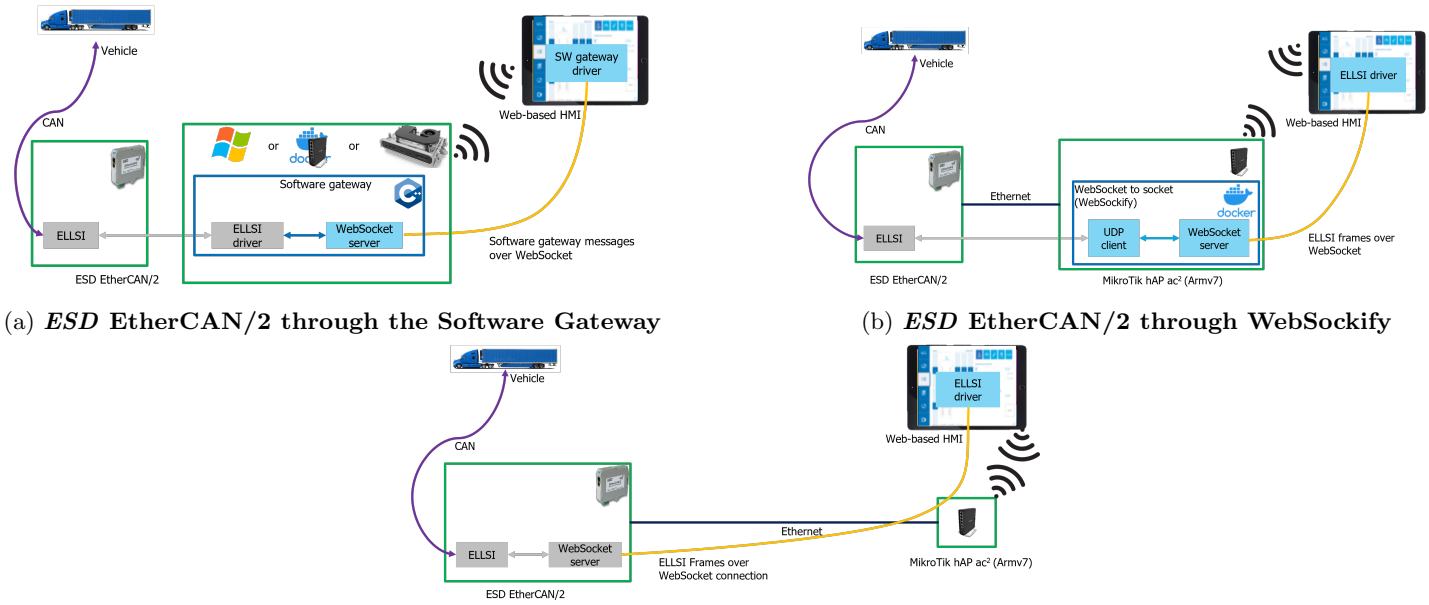
## *ESD* EtherCAN/2

The *ESD* EtherCAN/2 is a standalone box that has one Ethernet port and one CAN interface. It runs multiple protocols: it provides a UDP interface with its ELLSI protocol, and it also supports this protocol on a WebSocket interface. *ESD* provides a C and a JavaScript (JS) implementation. The protocol uses telegrams with control commands or one or more CAN messages. The EtherCAN/2 sends a UDP/WebSocket message every 10 millisecond (ms) to put multiple CAN messages into a single telegram.

When using the *ESD* EtherCAN/2, there are three main possible architectures with the new HMI system. These are shown in Figure A.1.

The first network (Figure A.1a) uses the Software Gateway on any capable hardware and lets it connect to the EtherCAN/2. The Software Gateway can be on any hardware because the EtherCAN/2 can be reached from the Ethernet network using the C driver. Using the Software Gateway provides the added functionality of the Software Gateway, such as rate filtering, better filtering and broadcasting. It also lowers the load on the EtherCAN/2 when there are multiple HMIs in the system. The EtherCAN/2 then only needs to communicate with a single socket client and not multiple WebSocket clients.

The second option (Figure A.1b) is connecting to the EtherCAN/2 through WebSockify, with the *ESD* ELLSI JS driver on the HMI. This driver usually




(a) ***ESD* EtherCAN/2 through the Software Gateway**

(b) ***ESD* EtherCAN/2 through WebSockify**

(c) ***ESD* EtherCAN/2 directly**

Figure A.1: **Different architectures for the *ESD* EtherCAN/2**

connects directly to the EtherCAN/2, but as the ELLSI protocol is used in WebSocket messages, the WebSocket layer can be stripped, and it still works. This shifts the load for providing the WebSocket interface from the EtherCAN/2 to the WebSockify runner (the router). This option would only be recommended when the EtherCAN/2 is not able to provide multiple connections to a large number of HMIs, and it is not possible to use the Software Gateway. It does not add any extra functionality.

The last setup (Figure A.1c) uses the new HMI JS driver system, with the *ESD* ELLSI JS driver and connects to the EtherCAN/2 directly from the HMI. This is the same as the current setup shown in Chapter 3, but using the new JS gateway system. This makes it possible to change hardware by only changing the configuration on the HMI and not completely switching drivers in the code.

The last option is recommended For a simple HMI system with a few (¡3) HMIs and no inter-HMI communication, as this is the easiest and does not require extra hardware or setup. For more complex systems with more HMIs and inter-HMI, the first option is recommended, as this will take most of the load off the EtherCAN/2 and could be more network efficient.

## Vector XL

Vector Informatik produces hardware and software for automotive companies. One of their products is the VN7640, a hardware interface with CAN (FD), LIN and FlexRay that can be connected through an Ethernet connection. Vector does supply their own software for sending/receiving data to those buses, and it is even possible to make simple graphical user interfaces (GUIs) with their

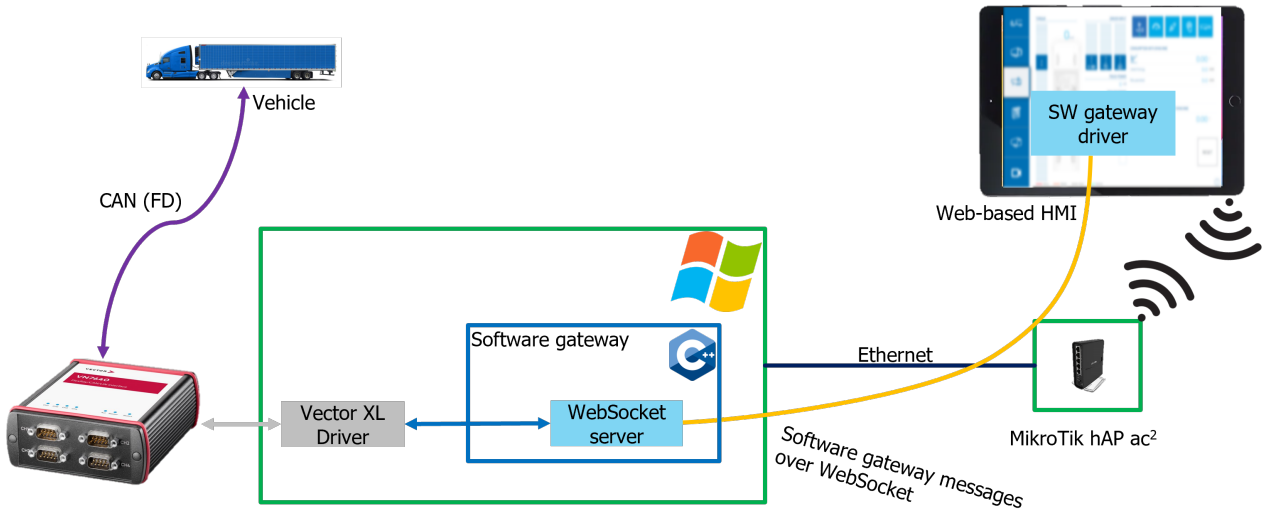



Figure A.2: **VectorXL architecture**

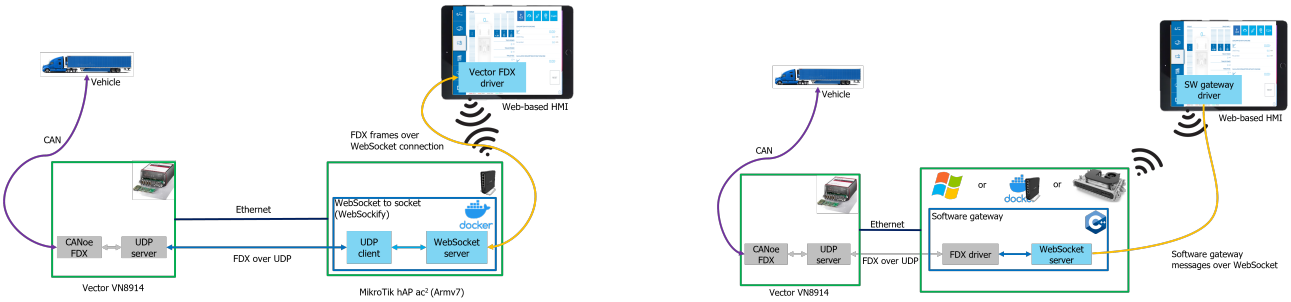
software. It is possible to talk to the hardware interfaces with their Vector XL driver. This is a proprietary C driver compiled to a dll file for Windows. This way it is required to use Windows for the Software Gateway. The Vector XL driver does not communicate directly with the hardware interface, but uses an intermediate system where all the needed hardware interfaces are connected and then it is possible to control a channel, a single bus of a hardware interface, and it makes it possible to have multiple applications connect to the same channel.

This means that before using the Software Gateway, the Vector XL driver needs to be installed and configured. The MVP could embed the driver installation in the installer, however this would increase the 1.5 MB installer with 1.47 GB and this might not be allowed by their license.

The Vector XL library was added to the MVP and is only included when built for Windows. The configuration checks are always compiled though, even when compiling for Linux, to be able to check configurations of Windows Software Gateways that the Linux Software Gateway might connect to and upload configurations to. The same is done for SocketCAN configurations on Windows Software Gateways.

In Figure A.2 the possible system for using Vector XL devices is shown. Vector hardware interfaces are often used for development and are often already incorporated in an Innovation Vehicle. The only thing that needs to be added to the system is a Windows computer for the Software Gateway. There are small computers, like Next Unit of Computings (NUCs), that can fit easily in a vehicle and are cheaper than any hardware interface possible with the MVP. With the Software Gateway, the HMI is able to connect to this interface and communicate with a lot of buses in the vehicle.

When using Vector devices supported by the VectorXL driver, like the VN7640 and the CanCaseXL, the Vector XL Software Gateway driver can be used. This driver only works on Windows, as Vector only supports their driver on this operating system.




(a) **FDX system using WebSockify** (b) **FDX system using a Software Gateway**

Figure A.3: **Possible FDX architectures**

# Vector FDX

Vector also produces other hardware, like the VN8914. This is a standalone computer with multiple buses, like CAN (FD), LIN and FlexRay. It is possible to connect to this with the Vector XL driver, but this requires a Windows computer. The VN8914 also supports the Fast Data Exchange (FDX) protocol. This is a protocol that uses UDP messages with one or more commands in a datagram. These commands include DataRequest, DataExchange and FreeRunningRequest. All these commands are used with groups. These groups are defined beforehand in a FDX file and include the ID of each group and what signals should be in that group. This configuration should be the same on the client and the VN8914. The client then can request a group with the DataRequest command or request that group to be sent cyclically using the FreeRunningRequest. The VN8914 then will send the data in a DataExchange command to the client.

The FDX protocol is quite good documented and Vector provides a C++ reference implementation. This implementation is used for the Software Gateway and using the documentation, a driver is made in JS that uses WebSockify to convert the UDP datagrams to WebSocket messages and vice-versa.

The two possible system layouts are shown in Figure A.3. Figure A.3a shows the architecture with a WebSockify running on the router to convert the WebSocket messages to UDP frames. This system is recommended when there are not too much HMIs in the system, as they each connect to the VN8914, resulting in duplicate data from the VN8914 to the router.

Figure A.3b shows the VN8914 used with the Software Gateway. The Software Gateway can be run on any capable hardware, as there is no requirement for Windows when using the FDX protocol. When there are more HMIs connected, this system is recommended to take load off from the VN8914 and reduce network traffic.

# ZF ProConnect

ZF develops, besides vehicle transmissions and car parts, also electronic parts for the automotive industry, like the ZF ProConnect. This is an automotive computer with CAN (FD), LIN and other buses, running an NXP i.MX8 processor.

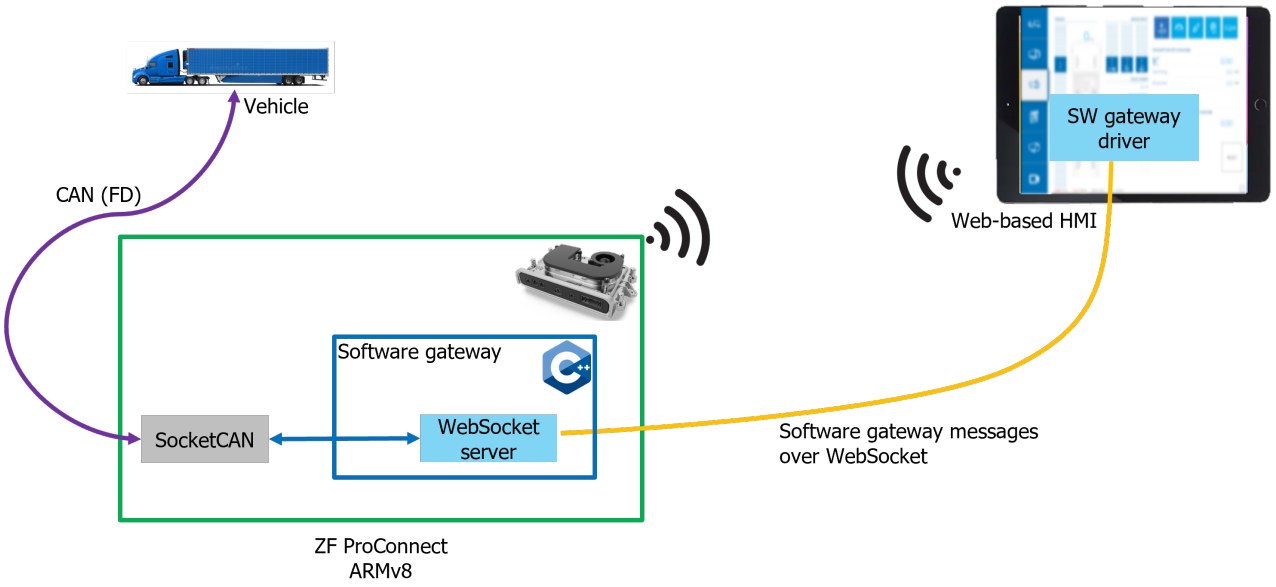



Figure A.4: **ZF ProConnect architecture**

It runs a bare Linux distribution, and with a dedicated toolkit, it is possible to compile C/C++ for it. It supports SocketCAN, so it can be used easily with the Software Gateway. A simple ZF ProConnect system setup is shown in Figure A.4. The ProConnect can also be a Wi-Fi access point, removing the requirement for a router in simple systems.

The SocketCAN system makes it easy to use and with the help of the SocketCAN-cpp library with added support for the Broadcast Manager (BCM), it is one of the smallest drivers. SocketCAN makes CAN (FD) buses available as network interfaces and thus these buses can be used with normal Socket programming. With SocketCAN it is easy to change hardware interfaces and it is possible to make virtual CAN (FD) buses to simulate buses and to communicate with CAN (FD) applications on the same computer. The Software Gateway also supports this and for broadcasting switches the broadcasting algorithm to use SocketCAN's BCM system. Delegating the cyclic sending to the kernel instead of the application should make it more stable and less CPU-intensive.

# Appendix B

# Broadcasting algorithm

**Algorithm 1** Broadcasting algorithm

```
delay ← gcd(messages.interval)
for message in messages do
    message.divisor ← message.interval/delay
end for
maxLoop ← lcm(messages.divisor)
loopCounter ← 0
loop
    for message in messages do
        if loopCounter%message.divisor is 0 then
            sendMessage(message)
        end if
    end for
    loopCounter ← loopCounter + 1
    if loopCounter is maxLoop then
        loopCounter ← 0
    end if
    sleep(delay)
end loop
```

The algorithm first calculates the greatest common divisor (GCD) of the intervals. This is the delay that fits nicely in all the intervals. Then a divisor is calculated per message. This is how often the delay must be done between each cycle for that message. Then a least common multiple (LCM) is calculated. This is the number of times the delay must be done such that all the intervals fit nicely and the loop counter can be reset. This prevents integer overflow.

The same algorithm can be used for this broadcast timeout checking, but instead of sending the message, it can check a list to check that the message was received in that interval.

# Appendix C

# Rate limit algorithms

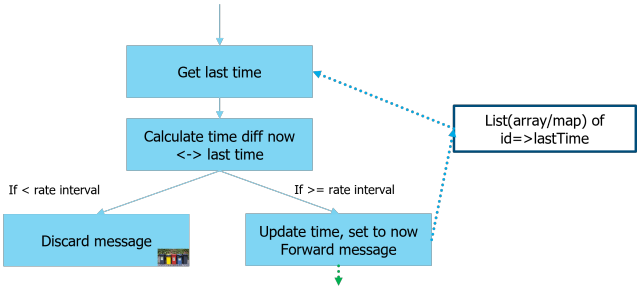
In Figure C.1, some algorithms for rate limiting are shown. Figure C.1a shows an algorithm that stores the time the message has been sent last, and when the time since the last message is more than the rate interval, the message is forwarded, and the storage is updated. This is the most basic algorithm, but it is a bit more expensive on the calculations, as getting and calculating the time is more expensive.

In Figure C.1b, a nearly similar algorithm is shown, but instead of storing the last sent time, a boolean, whether it was sent in the last time slice, is stored. Then every rate limit interval, the list of booleans is cleared, and the first message of that ID is forwarded again. This requires an extra thread but is cheaper per received message, as it is only a lookup instead of an additional time calculation.

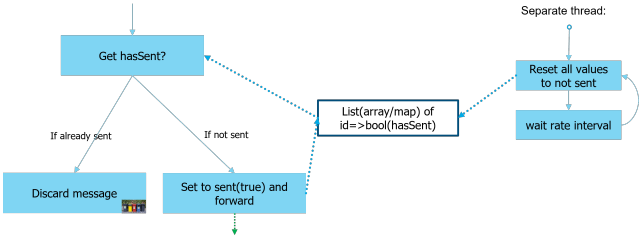
A counting algorithm is shown in Figure C.1c. This algorithm only forwards every $N^{th}$ message with a reset at every rate interval. This interval is needed to continue forwarding each message with a low rate compared to faster messages, which only need every $N^{th}$ message. This algorithm is probably as expensive as the time-sliced rate limit algorithm but can have a longer rate interval, resulting in fewer thread wake-ups and writes and fewer message storms at each time slice start.

A queued approach is shown in Figure C.1d, where a queue of messages is made each time slice and then sent all at once. The queue should only contain unique message IDs to keep the last received message or discard new messages if they already exist in the list. This queued algorithm will take more memory than the other algorithms as it needs to store the messages with their data instead of a flag or time point. Meanwhile, it will forward all the messages at the same time, which could result in better WebSocket message usage, as it could fill it up more.

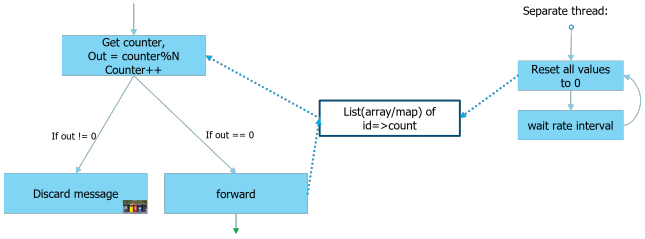
The time slice rate limiting algorithm of Figure C.1b is probably the optimal algorithm, as it does not need as much storage as the independent time triggered and queued algorithms and is stricter on timing than the counted algorithm.



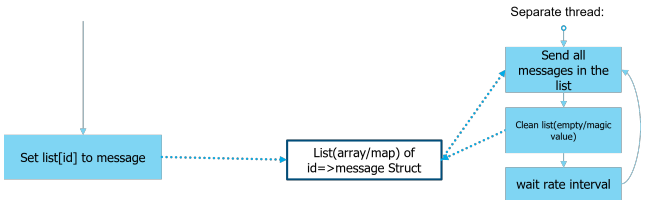

(a) **Independent Time Triggered rate limiting**




(b) **Time slice rate limiting**




(c) **Counted/Decimation rate limiting**




(d) **Queued time triggered rate limiting**

Figure C.1: **Different algorithms for rate limiting**

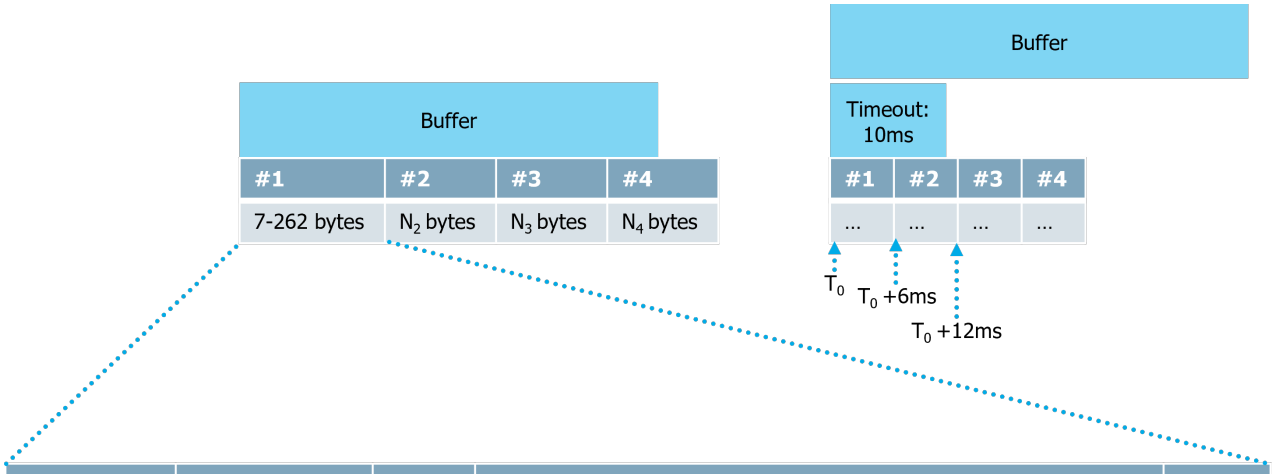
# Appendix D

# Bus Messages

Figure D.1 proposes how to send the bus messages in WebSocket messages. The flags byte is there to signal special cases, like broadcasting. Then the data length is sent with the bus ID. This ID is unique for each bus in the config and keeps track of which bus the data is meant for or came from. Subsequently, the message ID is sent and finally, the data. The length field defines the length of the data. This proposal will fit any CAN, CAN FD, LIN, and FlexRay message. CAN XL messages can be longer and will not fit this proposal type. Extending the length field to two bytes would also make those messages fit.

The top (left) in Figure D.1 shows that multiple messages fit in a buffer, but not all. Message 4 does not fit and will be stored in a buffer for the next WebSocket message.

On the right is shown when a message is sent every six ms with a timeout of ten ms with a very large buffer compared to the message size. The first two messages will be packed in a single WebSocket message, where message 3 arrived late to the timeout and will be put in the next message, even though the buffer was not full.




| Flags<br>1 byte | Length of data<br>1 byte | busID<br>1 byte | ID<br>4 bytes | | | | Data<br>... |
|---|---|---|---|---|---|---|---|
| 0b0000_000x<br>For broadcast | 0-255 | 0-255 | Lowest byte | Id>>8 | Id>>16 | Highest byte<br>&0b1000_0000 for extended | 0-255<br>bytes |

Figure D.1: **Message definition and buffer and timeout examples**