UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

TFG del Grado en Ingeniería Informática

Developing a CAN Ethernet Gateway for Software Defined Networks in Future Cars

Presented by Yunus Ülker
in the Universidad de Burgos — June 6, 2022
Tutor: Juan Jose Rodriguez

UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

D. Juan Jose Rodríguez Diez, profesor del departamento de Ingeniería Informática, área de Lenguajes y Sistemas Informáticos.

Expone:

Que el alumno D. Yunus Ülker, con DNI L2RFP18KZ, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado título de TFG.

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, June 6, 2022

Vº. Bº. del Tutor:

Firmado por RODRIGUEZ DIEZ JUAN JOSE con un certificado emitido por AC FNMT Usuarios

D. Juan Jose Rodríguez Diez

Vº. Bº. del co-tutor:

D. nombre co-tutor

Abstract

The raise of new technologies for in-vehicle networks, as for example, Highly Automated Driving, comes with a high amount of demands. The traditional in-vehicle networks cannot meet all of them, which gives the motivation to search for alternatives. A promising solution is to use high-speed ethernet backbone networks to connect various traditional networks such as CAN buses.

This work provides a gateway solution between CAN buses and ethernet networks. The specialty of this gateway is that it acts like an SDN network device instead of having a local decision making logic. After a deep evaluation the resulting gateway prototype is deployed in the experimental setup of the CoRE research group.

Keywords

Gateway, In-Vehicle network (IVN), Software Defined Networking (SDN), Ethernet, CAN, Real-time communication, OpenFlow

Contents

List of Figures

List of Tables

Introduction

In-vehicle networks are needed to connect sensors, actuators and Electronic Control Units. Traditional in-vehicle networks were statically configured and deployed in the manufacturing process. New technologies such as Advanced Driver Assistance Systems, Highly Automated Driving and Over-the-Air-Updates have higher demands than the traditional solutions can provide. This problem motivated the study of using ethernet as a highspeed backbone network connecting different domains [7]. Further studies are analyzing the use of Software Defined Networking (SDN) to make the ethernet networks more flexible and secure [9].

This work contributes to the studies about the use of SDN ethernet backbones networks. Multiple transport protocols are used for in-vehicle networks and need to pass a gateway to be sent over the ethernet backbone network. Traditional gateways are implemented with one specific translation strategy and a static local configuration. Since the idea of SDN is to make network devices become simple forwarding devices, and not make their own decisions, traditional gateways can not be seen as a part of an SDN. The idea of this work is to develop a prototype gateway being part of an SDN, with all the resulting functionalities and advantages.

Project Objectives

The project objective is to design, implement and evaluate a CAN ethernet gateway solution. The special challenge of this gateway is, that it shall not have its own control logic. Instead, it should be programmable via protocols of the SDN family. An extensive performance evaluation will be carried out and when obtaining the desired results, the prototype will be deployed in the experimental setup of the CoRE research group[8].

The following images demonstrate the traditional architecture of gateways, shown in figure 2.1, and the used gateway architecture for this work, shown in figure 2.2.

Outsourcing the control logic to an SDN controller creates the need of configuring the controller with the desired forwarding behavior for the gateways. In this work this is done for ONOS as the SDN controller.

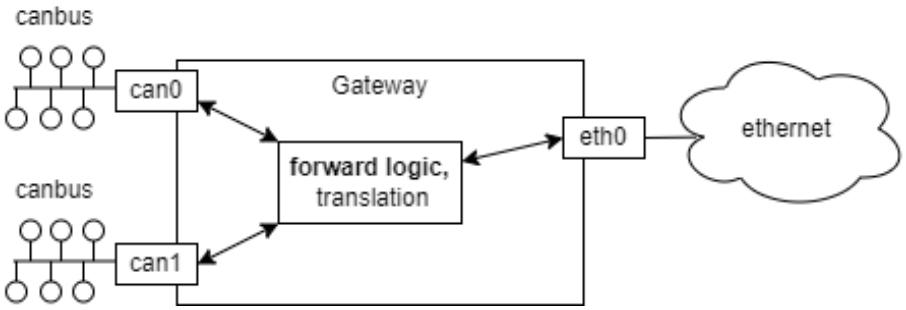


Figure 2.1: Traditional gateway architecture, forward logic inside the gateway

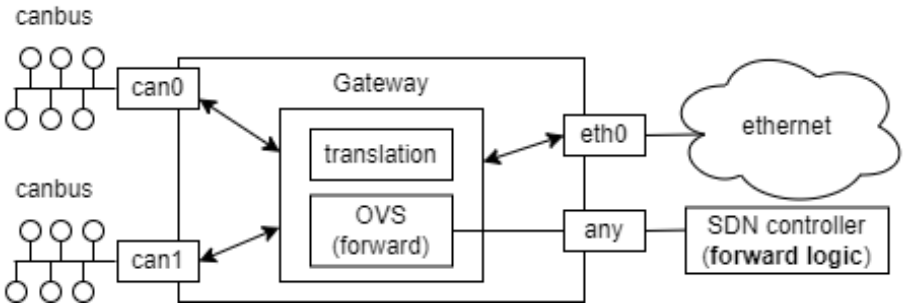


Figure 2.2: Gateway architecture used in this work, forward logic outside the gateway

Theoretical Concepts

3.1 Controller Area Network

CAN was developed by the Robert Bosch GmbH and became standardized in 1993 with ISO 11898 allowing a bandwidth up to 1Mbit using only the *standard* 11-bit identifier[10]. In 1995 this standard got an amendment including the *extended frame format* supporting a 29-bit identifier[11] supporting the same 0-8 bytes as data field. Since the CAN bus is broadcast-based it is not possible to address a specific destination or to know the origin of a received message. Neither it offers an authentification process.

The Controller Area Network (CAN) architecture is used in most automotive systems[14]. The CAN bus is a multi-master serial bus system with all messages being broadcasted. Dominant and recessive bits assure that higher prioritized messages suppress others without any additional regulations needed. Every sending process of an electronic control unit (ECU) starts with a start-of-frame (SOF) bit followed by a unique identifier code. This identifier establishes the priority of the message. If two ECUs are trying to send their identifier code a conflict will occur. The lower priority-based ECU will notice that his sent recessive bit (logical one) was overwritten by a dominant bit (logical 0) of a higher prioritized message. This leads to aborting the sending process for the lower prioritized message. Messages always end with a CRC checksum for error detection and a recessive set ack-Bit, which has to be overwritten with a dominant bit from the receiver to acknowledge the error-free receiving[2].

This architecture is optimal for sending many short messages, for example, temperature sensor data or the angle of a steering wheel. Mainly because of the possibility to prioritize safety-critical messages, data consistency through

the broadcast architecture and the low costs, which result from a small number of cables and little planning effort.

To deal with the growing amount of ECUs a typical practice is to connect and combine different CAN domains. Often this is realized through backbone networks based on ethernet with the need of gateways. Ethernet without additional extensions is not supporting priorities and works with best-effort delivery. Best-effort delivery is not able to accomplish the requirements of safety-critical communication, which is necessary for automotive networks.

3.2 Real-time Ethernet

Many scenarios require real-time communication. In automotive networks, for example, an airbag always needs to receive the triggering data in time to work safely. Since this is about safety it is necessary to always guarantee it. This is called a hard real-time requirement which makes it a safety-critical system. Because of those scenarios real-time extensions for ethernet are needed to use ethernet in a vehicle network. The most extended solutions for the use of ethernet in safety-critical systems are Time-triggered Ethernet (TTE) and Time-Sensitive Networking (TSN)[20].

TSN supports sending time-triggered as TTE does, but also supports sending priority based. As TSN is integrated into the IEEE 802.1Q standard, which enhances ethernet frames with a priority field, a VLAN identifier, and a drop eligible bit. Plenty of strategies are available to deal with prioritized message sending, called shaping in the context of TSN. The simplest shaper to comprehend is always sending the highest priority frame first.

Here the worst case would be a high prioritized frame arriving exactly after starting to send a maximal sized frame, causing the high prioritized frame to wait until the transmission completes. To reduce that maximal delay *frame preemption* defined by IEEE 802.Qbu [20] is introduced, reducing the maximum delay to the time that the preemption of a frame needs, instead of waiting to complete the transmission. The only way to reduce this dynamic delay even more, is to not even start sending traffic before high prioritized frames arrive, which is the idea of time-triggered sending.

To achieve time-triggered sending TSN and TTE are based on synchronized timers between the different networking devices and the use of Time-Division Multiple Access (TDMA). Both of them use off-line schedule tables to send time-triggered (TT) data. Those schedules repeat cyclically. TTE divides whole cycles, called cluster cycles, into sub-cycles, called inte-

gration cycles [13]. The clusters cycles just repeats and whenever there is no TT data sent, best effort (BE) data is possibly transmitted. A simple example with two integration cycles for each cluster cycle is shown in figure 3.1.

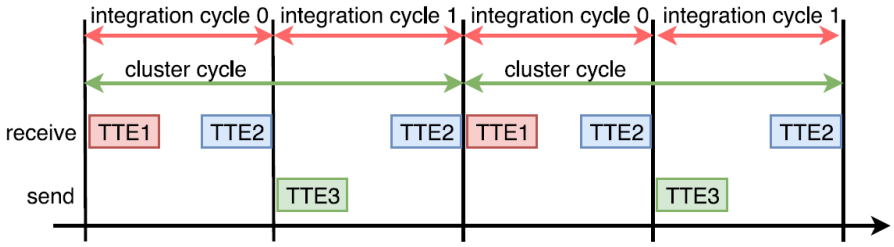


Figure 3.1: TTE, simple schedule example [13]

TSN splits the cycles into slots with assigned priorities as exemplarily shown in 3.2.

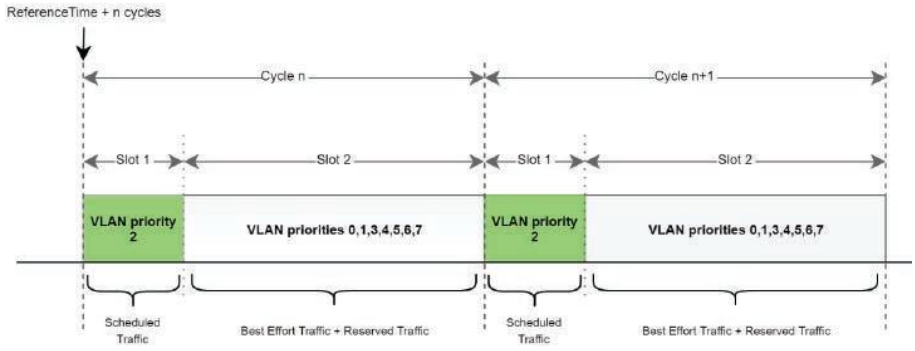


Figure 3.2: TSN, simple schedule example [5]

Even though predefined timeslots are minimalizing the delay and jitter, they have the disadvantage of needing an offline schedule and synchronized timers on the networking devices. Depending on the requirements it is often enough to use TSN with shaping algorithms not having those restrictions. But these are basically the options for real-time environments using ethernet.

3.3 Security for In-Vehicle Networks

In modern vehicles the five most widely used in-vehicle networks systems are LIN (Local Interconnection Network), CAN (Controller Area Network),

FlexRay, Ethernet, and MOST (Media Oriented Systems Transport)[19]. As in many old protocols, the security aspect was not extensively considered inventing. CAN, for example, is natively not offering any form of access control, authentication or encryption [18]. In the past vehicles were rarely connected, so this lack of security did not imply a big security threat, because a possible attacker needed physical access.

The evolvement of new technologies like Advanced Driver Assistance Systems and Highly Automated Driving makes the interest in Connected and Automated Vehicles grow. While connecting cars to the outside world promises enhanced functionality and flexibility, the security lacks of long-time used protocols become a relevant threat. To counteract there are many protocol-specific extensions to close different kinds of security gaps. Analyzing in detail and evaluating many different extensions, for numerous network technologies, is out of scope for this work. Those attempts mainly focus on giving more security for certain communication channels.

It could be possible to introduce a central unit that has knowledge of traffic over the whole system. This would allow solutions across different protocols. After the up-showing threat via the connection to the outside world, the biggest potential threat is coming from newly added devices or applications [9]. With a central unit there could be one standardized authentication process improving security, flexibility and configuration effort. The global view of a central unit also facilitates recognizing errors or possible attacks using anomaly detection strategies. There are plenty of different strategies and algorithms for anomaly detection, but the basic idea is to compare the actual traffic with the expected one. The recognition of errors and attacks is a necessary step to enable fail-operational states.

3.4 Software Defined Networking

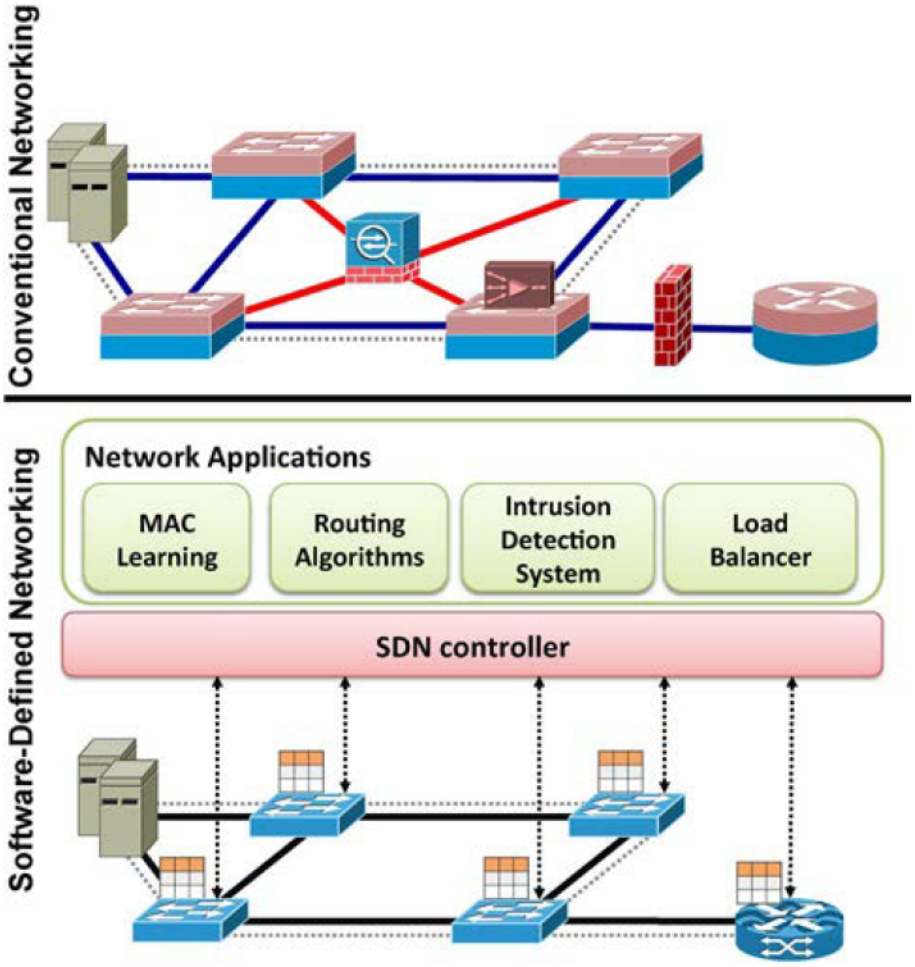


Figure 3.3: Comparing SDN to conventional networking.[12]

The Open Networking Foundation (ONF) defines SDN as the physical separation of *the network control* plane from *the forwarding plane*, and where one control plane is in control of several devices. "Software-defined networking: A comprehensive survey"[12] defines an SDN as a network architecture with the following four characteristics. The control and data plans are separated, making networking devices simple forwarding elements, that are only in charge of the data plane. Instead of using the destination of a package for the forwarding decision, the control plane decides flow-based. Flows are filter criteria to match packets via their field values and a set of instructions to apply to them (e.g. to forward them over a specific port). The control logic is moved to an external and logical centralized entity,

called the SDN controller. The network needs to be programmable through applications running on the SDN controller.

Even when there are many different definitions available for SDN, figure 3.3 is already giving a good understanding of the architecture. SDN controllers always offer two different APIs. The southbound API communicates with the forwarding devices and the northbound API to allow the development of network applications. This *separation of concern* gives many advantages in flexibility and security. Developing network applications or modifying network policies with high-level languages, using an API is way less error-prone and simplified compared to the traditional low-level device-specific configuration. Additionally, a control program could react to changes in the network. This functionality greatly benefits from the global view of the network.

When developing network applications with SDN, the southbound API behavior is similar to the behavior of well-standardized hardware drivers. The function of generating statistics by counting matched flow-entries, for example, is natively included in the most widely supported southbound API OpenFlow. This ability makes anomaly detection discussed in section 3.3 for in-vehicles networks interesting. Also, the possibly following fail-operational state can be set up using OpenFlow.

3.5 Gateway

Seo et al. describe a getaway as "an indispensable device to enable the seamless communication between heterogeneous networks." [15] With heterogeneous networks meaning networks using different protocols and media.

As exemplarily shown in figure 3.4, modern cars have many different networks. Depending on their demands, they use different protocols to assure, for example, low latency, high bandwidth, fail-tolerance or low costs.

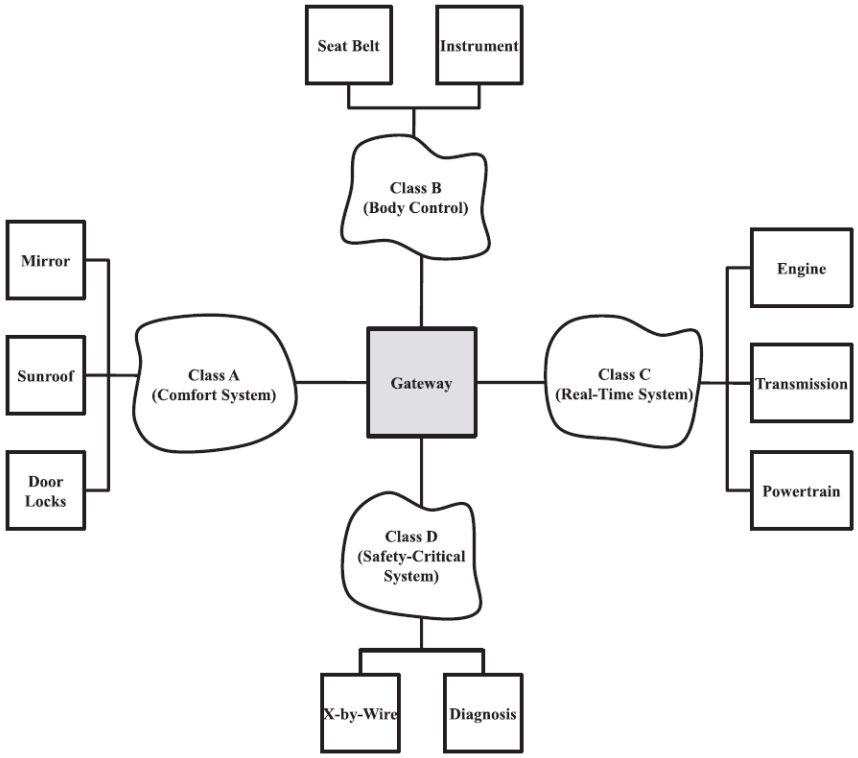


Figure 3.4: Examplary topology for an automotive gateway. [15]

There are several reasons which make gateways in the context of in-vehicle networks interesting. For example, connecting an ethernet backbone network via gateways to different buses to reduce the number of cables needed, lowering costs and weight.

Gateways for in-vehicle networks have many requirements, especially when they need to serve as hard real-time systems. They need to be reliable, even having in mind environmental influences like vibrations or temperature. Since this work is about researching an approach including developing a prototype and not about developing a deployable product, many of the final requirements are out of scope or are only slightly discussed. Protocol-specific translation strategies to enable more seamless communication between an SDN and different CAN busses, are the main focus in the context of this work.

3.6 Concept of the Approach of this Work

A gateway needs to translate one protocol into another. Both CAN and ethernet are based on frames. There are two different decisions to make. The smallest ethernet frame is still four times bigger than the biggest CAN frame. This difference causes the need for the first decision. Should the gateway translate one frame to one frame or rather encapsulate many CAN frames into one ethernet frame. Depending on that decision, it is possible to make the second one. Both protocols contain different information, which cannot be translated directly. For example, the biggest part of an ethernet frame is occupied by source and destination information, meanwhile CAN is broadcast oriented and does not provide any information about the destination nor the source. So it is necessary to choose how to translate one protocol into the other.

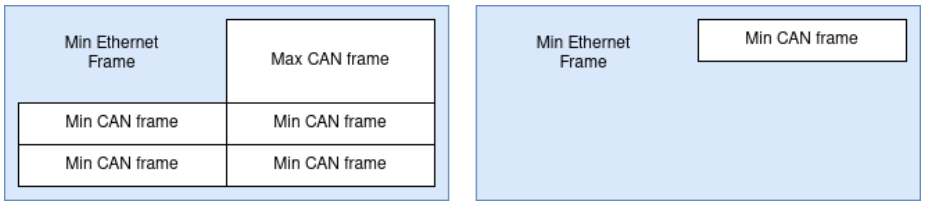


Figure 3.5: Visualizing the size differences of CAN and ethernet frames.

The strategy to translate frames one to one has the obvious disadvantage of large traffic overhead. To better understand the amount of overhead there is a simple example. In a point-to-point connection, one megabyte of traffic packed as maximally sized CAN frames would turn into four megabytes of traffic packed as ethernet frames. But considering the associated bandwidth of the different links, the big ethernet frame takes only 0,2% of the time the small CAN frames take to be transmitted, so this overhead is not a big problem.

If two CAN frames would be sent encapsulated into one ethernet frame instead, the traffic amount would get halved, but causing a delay for the first CAN frame since it will not get sent until the second frame arrives. This delay-to-overhead conflict is analyzed in many gateway research documents. In this work another aspect is relevant. Each CAN frame can have a different destination and the gateway does not know them. If CAN frames with different targets are encapsulated in one ethernet frame, another instance would need to receive, split them up and resend them again separately, resulting in even more traffic than directly sending frames mapped one-to-one.

Another argument to choose one-to-one mapping is that using SDN instead of a traditional networking, makes the traffic of the network transparent and analyzable. If many CAN frames are encapsulated into ethernet frames, SDN would not be capable of analyzing each CAN frame on its own.

Taking all those arguments into account, the author's choice of one-to-one mapping is comprehensible. The most important argument for choosing one-to-one mapping is to get the SDN-related benefits for the CAN traffic. Generally SDN controllers can read all fields from layer two to layer four, corresponding to the OSI layer model. Therefore, to get those SDN-related benefits, it is necessary to map CAN fields into fields lower than layer five, which means to not map the important information simply into the *data* field.

Which CAN field to map with which ethernet field is a design question. It is especially difficult because the ethernet is used for point-to-point communication and CAN is broadcast-oriented. The biggest part of an ethernet header is occupied by source and destination information, CAN frames do not provide any. The proposed solution to this question is shown in figure 3.6.

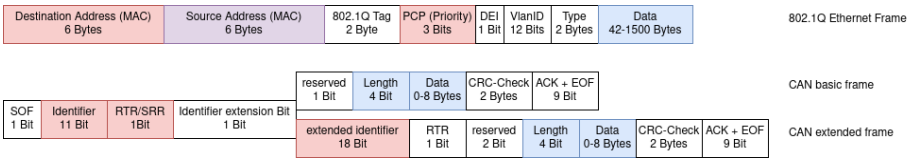


Figure 3.6: Shows the mapping of CAN and ethernet frame fields.

The destination MAC address is mapped to the CAN id field and vice versa. The source MAC is not mapped to any CAN field, but it is read and set by the gateway program to distinguish and represent the different connected CAN buses. This source MAC values need to be configured manually to assure network-wide uniqueness. An example of a CAN frame translated to an ethernet frame is shown in figure 3.7. As this section serves to show this concept, much information shown in the example like SSR flags are not explained in this part, but will be explained in detail in the Annex document.

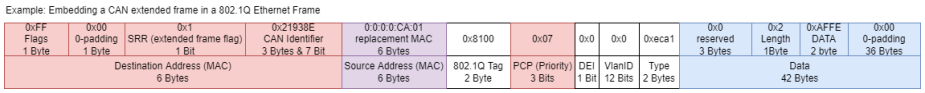


Figure 3.7: Example: A translated CAN extended frame as 802.1Q Ethernet Frame.

Even if SDN controllers are capable of interpreting and changing fields, this design follows the rules of the ethernet protocol. This is not only about sticking to conventions. The fact that the gateway behaves exactly as any other network device, makes it possible to use this gateway to connect any CAN bus to any ethernet network.

Tools and Technologies

4.1 Setup and Implementation

4.1.1 Open Network Operating System (ONOS)

The Open Networking Foundation describes ONOS as "the leading open source SDN controller for building next-generation SDN/NFV[1] solutions"[6]. The special about this gateway approach is that the gateway's routing logic is not part of the gateway itself. Instead, the gateway only receives routing rules via the southbound API of an SDN controller. The provided complexity, which differs with every SDN controller, is not exploited for this work and therefore the question of which SDN controller to choose would not matter. But since this work contributes to the CoRE research group, it is necessary to be compatible with their experimental setups, which use ONOS as an SDN controller. For real use, the companies would develop a vehicle-specific SDN controller anyway.

4.1.2 Open vSwitch (OVS)

Open vSwtich is an open-source virtual switch software and "is used in multiple products and runs in many large production environments (some very, very large). Each stable release is run through a regression suite of hundreds of system-level tests and thousands of unit tests."[17]

As the quote shows this software mainly targets big virtualized networks, but the most functionalities are not needed in the scope of this work, neither will they. The reason to use a virtual switch in this work, is that the gateway becomes controllable via standardized protocols, which allows controlling

[1] network functions virtualizations

the gateway as part of the SDN with all resulting benefits. To do so, one virtual switch is installed on each gateway having an interface to the ethernet network, including a route to the SDN controller, and for each CAN bus connected, a virtual ethernet port linked to a CAN bus via an instance of the gateway software.

4.1.3 cURL

ONOS, as chosen SDN controller for this work, provides three different interfaces. A web-based graphical user interface (GUI), a command-line interface (CLI), and a REST API. Curl is just one of many open-source tools available that enable communication with a REST API and could be replaced by another tool.

4.1.4 ip command from iproute2

Ip is the standard tool to show and manipulate routing, network devices, interfaces and tunnels under Linux. In modern Linux distributions, it is pre-installed as part of the iproute2 collection. Since this work strongly relies on network functionalities and uses a Linux distribution as OS, using *ip* is the modern recommended way to set the network devices up.

Installing the driver collection *SocketCAN* makes *ip* support CAN and virtual CAN buses and is therefore needed.

4.1.5 Sockets

Sockets are interfaces to network devices provided by the OS. They connect user processes with the network stack at different entry points. Instead of using the socket as interface to the transport layer, this work uses them as interface to the data link layer, which is the lowest software-defined layer corresponding to the OSI reference model. This principle is also known as raw socket programming or more specifically layer two raw sockets programming.

Therefore, it is possible to set any value into any field before sending ethernet frames, which is necessary to realize the CAN to ethernet translation. This is needed, because the design of the approach requires, for example, overwriting the mac address field with CAN specific data.

4.1.6 can-utils

Can-utils is a collection of open-source userspace command-line utilities and tools based on SocketCAN. The most relevant tools for this work are cansend, candump, cangen and canplay providing simple functionalities like sending, dumping, generating or sending dumped traffic cyclical. Alternatively to can-utils, there are some python tools with similar functionality available and it would be possible to implement an own solution.

4.1.7 tcpdump

Tcmpdump is a widely-used open source command-line tool to filter and dump traffic. Also the name might be confusing, tcpdump is not restricted to dump only TCP packages. In this work it is used to listen on different interfaces, filter frames layer two fields based and dump them adding timestamps. Again there are many alternative programs to accomplish this task.

4.1.8 Python

Performance-wise, python gateway implementations can not fulfil real-time requirements as analyzed in previous work by the CoRE group. To inspect the measured times obtained by the dumping tools and visualize them, it is still a simple and efficient solution. All the figures in the evaluation section 4.2 are drawn using the pyplot library. Python was also used to troubleshoot traffic logs, when unexpected behavior occurred conducting experiments.

4.1.9 Bash scripts

Bash scripts do not give more functionality than the command line itself but make many processes way easier. Instead of knowing every needed command with all required parameters in the right order, users can read the provided manuals from the scripts and execute them with a small amount of parameters.

Some of the scripts are generally helpful, for example, the one to create, link and configure the interfaces and the switch including the setup of the connection from the switch to the SDN controller with a static port mapping between host and OpenFlow ports. Others are only used for testing purposes, as for example, to conduct the experiment Microbenchmark Gateway 4.2.1.

4.1.10 Git

The website git-scm.com belongs to the Git community and describes Git as "a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency" [1].

The main purpose for the use of git was to develop the translater program. The translator program was developed as a branch of an internal git repository of the CoRE group. Another private git repository, hosted by GitHub, was used to synchronize all other files. For those files, cloud services such as OneDrive, could have been used as well.

4.1.11 Trac

"Trac is a minimalistic approach to web-based management of software projects." [16]. The CoRE research group uses Trac to host an internal Wiki service. Much information, for example, how to register a public key at the git-server to access the repository, are given there. Also, the network configurations used in the experimental setup of the in-vehicle network and many other guides, are documented there.

4.1.12 Microsoft Teams

Most of the communication between members of the CoRE research group is done via Microsoft Teams. Additional to meetings, multiple channels are used to, for example, alert others when turning off the experimental setup of the in-vehicle network.

4.2 Evaluation

4.2.1 Microbenchmark Gateway Virtual

Purpose

This experiment is chronologically the first one and serves to determine how much time the different components consume approximately. The general purpose of this experiment is to assure that the gateway approach can be used in real-time environments, but having in mind that the final prototype will behave differently. Additionally, the resulting time values may be considered to improve certain parts of the strategies or algorithms, if the gateway approach does not fulfil the requirements.

Monitoring software is used to measure the needed time to transmit packets and frames over the different interfaces. The network topology and the measurement points are shown in figure 4.1. The obtained measurements then can be analyzed stochastically and graphically.

Procedure

The base for the experiment is the network topology shown in figure 4.1. The CAN buses, the ethernet links and interfaces, the switch and the SDN controller are virtual and running locally. No interfaces are shown for the connection between switch and controller, since they communicate over the IP protocol and therefore, the host machines is able of transmitting their data without the need of additional interfaces. The idea is to send CAN frames to *canbus 1* and receive them on *canbus 2*. To do so, the CAN frames need to be transformed into ethernet packages, switched over the ethernet network and be transformed back into CAN frames. To achieve this, the *CAN-Translators* need to be started and connected to the switch and the CAN buses. The controller needs to configure the flow-tables of the switch to make the switch forward correctly.

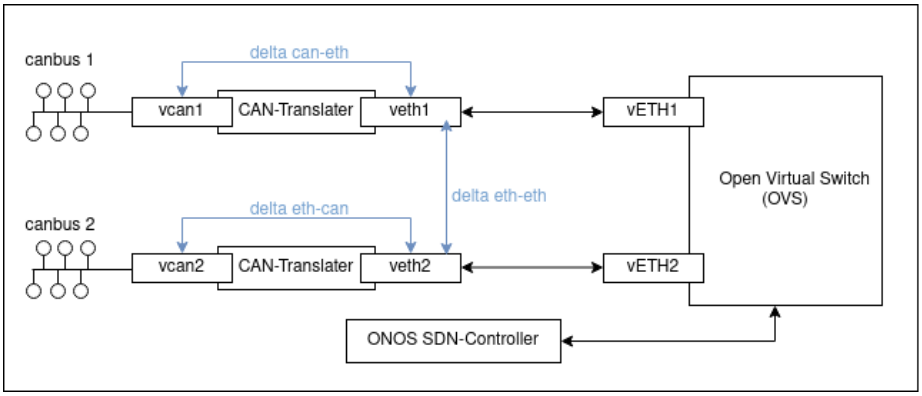


Figure 4.1: Network topology used in the virtual microbenchmark showing the three measurement points.

In this scenario it is enough to let the switch forward ethertype and income port-based. The OpenFlow port naming scheme is independent of the names which the forwarding devices use. So the mapping needs to be clear when creating the needed flow-rule. The mapping can be read via requests at runtime or be set configuring the switch, which makes it easier to set up or repeat experiments.

To make the switch behave accordingly, curl 4.1.3 and a JSON formatted OpenFlow entry are used to communicate with the *ONOS SDN-Cotroller* via its provided REST-API.

After setting up the network to be ready to transmit CAN frames from *canbus 1* to *canbus 2*, it is necessary to start dumping received traffic with timestamps on four interfaces as shown in figure 4.1. To dump the traffic of the CAN interfaces the tool candump 4.1.6 is used. For the ethernet interfaces tcpdump 4.1.7 with an ethertype filter is used. Since everything runs on the same host machine, there are no time synchronization difficulties and the difference between the timestamps can be used, to determine the needed time to traverse from one interface to the other.

Experiment Name	Figure	Delay in ms	Amount of Frames
g0n10000	4.4	0	10.000
g1n1000	4.2	1	1.000
g5n1000	4.3	5	1.000
g1n100000	4.5, 4.6	1	100.000

Table 4.1: List of used configurations for the experiment showing the given parameter to cangen

To meassure time in different scenarious cangen 4.1.6 with different parameters is used. Since the minimum ethernet frame size is bigger than a CAN frame, changing the CAN frame size via those parameters was not done. Only the parameters to set a delay, to wait after generating one CAN frame, and the amount after which to stop, were used.

All used configurations for the experiment are shown in table 4.1. Since the bitrate of virtual CAN interfaces depends only on the processor, which is shared with all other components, experimenting with this would not be meaningful. That understanding makes the provided parameter, which is only accurate to milliseconds, meet the requirements for this experiment. The results are plotted and shown as figures using python.

The most of the used commands can be read in the submitted script *virtualbenchmark.sh*.

Results

Before starting to analyze and possibly misinterpret obtained results, it is important to be aware of the testing environment. This experiment is

conducted on a personal computer with an i7 processor and mainly virtual hardware, so it is not possible to guarantee the same timing behavior on the final prototype.

It is also important to mention that the used time is not a constant and depends on many factors. In the most cases, the amount of traffic makes the biggest impact on the time measurements in a network. Virtual hardware additionally depends on getting computational time of the CPU from the OS. So, the CPU of the personal computer will cause faster working virtual hardware than the final prototype will have due to performance difference. On the other hand, it is expected that the number of background tasks will cause bigger impacts on the measurements than on the prototype.

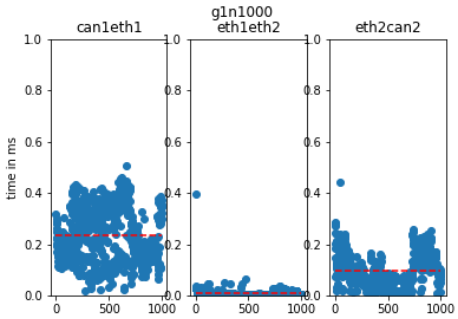


Figure 4.2: g1n1000

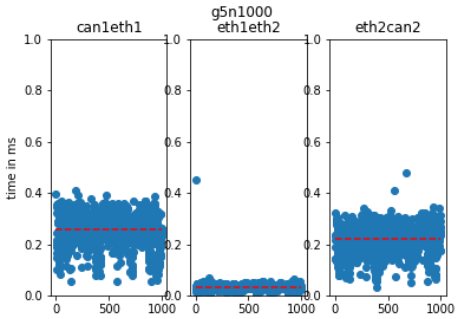


Figure 4.3: g5n1000

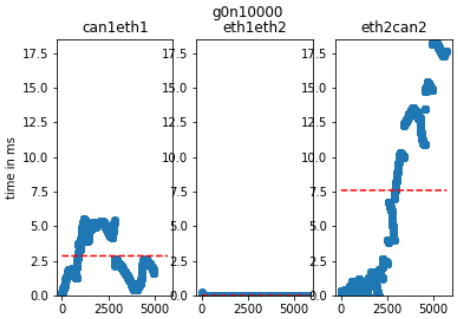


Figure 4.4: g0n10000

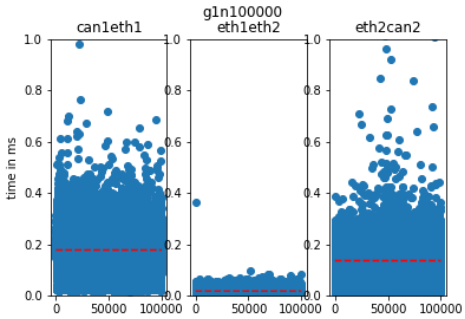


Figure 4.5: g1n100000

Configuration *g0n10000*, which sends without any delay resulted in a nearly ten times higher bitrate on the CAN bus, than the maximum bandwidth of the physical CAN bus would allow. Even though, this makes the measured times unusable, it is interesting to see the general behavior.

The gateways implement queuing algorithm to deal better with burst traffic, especially from fast ethernet to slow CAN. Instead of losing frames the needed time to transform ethernet frames to CAN frames raised to a thousandfold. Analyzing the queue size and limiting the amount of traffic being able to be switched to a gateway, needs to be done to avoid such a situation and to fulfil hard real-time requirements. But as mentioned above the environment of this experiment is explicitly not allowing to focus on those requirements.

This and the restriction to pass the delay in milliseconds leads to the choice of one millisecond as the minimal usable delay between each frame, as evaluated in configuration *g1n1000*. Configuration *g5n1000* has a five times higher delay parameter, which means that it sends five times less data in the same amount of time. Comparing the measurements of *g1n1000* and *g5n1000* it might be surprising that the configuration with higher delay transmits slower. This approves the factor of randomness, which cannot be avoided conducting experiments on a personal computer that does not have a steady process schedule. Sending 100 times more frames as done in *g1n100000* will help to reduce the effects of randomness. That amount of sample values makes it also possible to demonstrate the relation between outliers and scheduling issues as clearly demonstrated in figure 4.6, where all outliers can be located around 50.000 and 90.000 samples.

Since those are avoidable by choosing a different OS or simply having less -to this project unrelated- applications running in the background, they can be neglected in a worse case analysis.

With the foregoing discussion configuration *g1n100000* with a delay of one millisecond and an amount of 100.000 frames sent, is evaluated as the best configuration to use for the evaluation.

The translation, including the sending process, of a CAN frame as an ethernet packet can be found mainly between 0.02 and 0.45 milliseconds with an average translation time of 0.177 milliseconds. Adding the average switching time from ethernet to ethernet with 0.017 milliseconds and the average translation time from ethernet to CAN with 0.136 milliseconds, the transmission of a frame from one bus to the other needs 0.330 milliseconds on average.

To make an approximate worst-case analysis, the extreme outliers can be ignored as mentioned above. The slowest measured times for translating CAN to ethernet, switching ethernet to ethernet and translating ethernet to CAN are about 0.9, 0.15 and 0.7 and would cause a transmission time of 1.75 milliseconds.

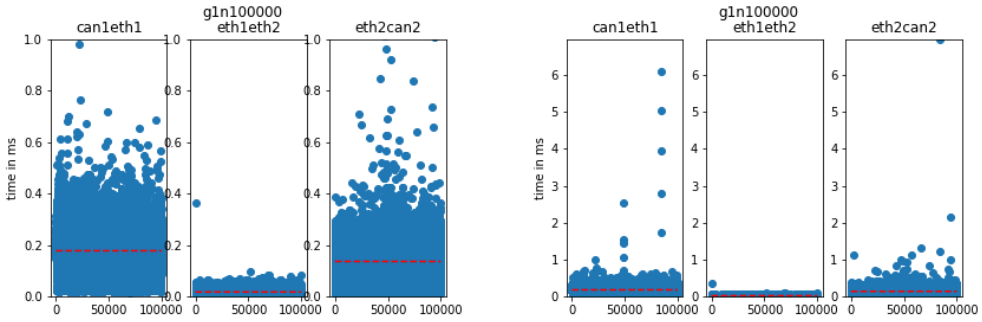


Figure 4.6: g1n100000 plotted twice. Once with scaling the y-axis to a maximum of 1ms and once showing all points.

Without guaranteeing the same speed using embedded computationally weak computers, the measurements confirm the potential to deploy this approach in a real-time environment and that the transforming takes more time than the switching.

4.2.2 Microbenchmark Gateway using Hardware 1

Purpose

The general objective is to test the performance in a non-virtual environment using an embedded computer and real CAN buses. This assures that the estimation of the performance is closer to a possible real deployment in the future. Also if the obtained results are good enough, this will allow to try replacing the existing gateway solution in the experimental setup of the CoRE group as the next experiment.

Procedure

The gateway is set up on a Raspberry Pi 4. Instead of using an external SDN controller, the needed flow-entries are installed manually via terminal. In total three computers, two CAN buses and one ethernet link are used. All computers are connected to a LAN via WiFi, which is independent of the CAN-ethernet network. The first computer is the raspberry working as the gateway, being connected to both CAN buses and the ethernet link. Another computer, working as the monitor, is also connected to these three links observing incoming traffic and saving timestamps. The third computer is used as the controller and is connected to one CAN bus to generate traffic. This computer is also connected to the others via SSH to execute

commands remotely and allow the use of scripts to automatize experiments. The described topology is shown in figure 4.7.

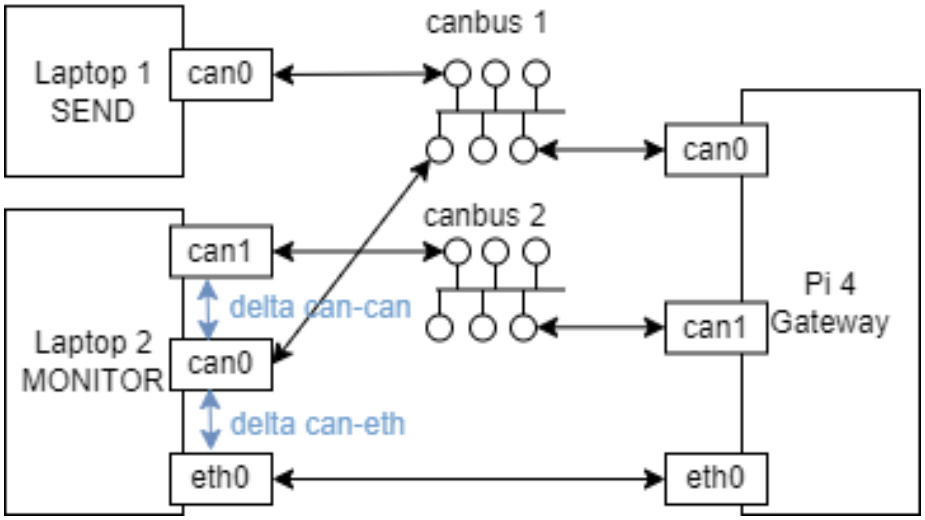


Figure 4.7: Network topology used in the microbenchmark using hardware shpwing the two measurement points.

To obtain a better understanding of the obtained performance, the experiments will always be conducted using the gateway approach, developed and presented by this work, and a solution similar to the existing solution used by the CoRE group. This solution does not include forward logic, instead, it forwards all traffic over the same interface given as start parameter. Also, the performance for different bandwidths and different traffic is analyzed. Therefore, the controller-computer generates CAN frames with different sizes of payload and changes the bitrate configuration of the CAN buses.

For each experiment the controller installs flow-entries to the virtual switch of the gateway to forward the traffic over the ethernet or CAN port. Then the translator program, developed for this work or the translator program similar to the one used by the CoRE group, needs to be started. The chosen bitrate for the CAN buses needs to be set on each computer for each connected CAN bus. The monitoring processes needs to be started for each interface to monitor. The controller generates equal CAN frames varying in their size for each experiment and sends them to one CAN bus. The monitor and the gateway receive them. The monitor saves the arrival time into a file. The gateway forwards them accordingly to the configuration over his ethernet port or over his CAN port. The monitor receives the forwarded traffic and saves the arrival time into a file.

For each configuration traffic will be generated with full utilization of the CAN bus for 10 seconds.

Results

Different values for three bandwidths, three frame sizes, two destination interfaces, and two different translator programs make 36 experiments. Figure 4.8 visualizes that amount of results as a violinplot, showing minima, mean, maxima and the distribution. Lines with "OVS" in the name mark that the translator program proposed by this work was used.

The most notable result are the big maxima values for mostly the OVS-cancan-1000kBit/s experiment series, but also the can-eth-1000kbit/s-payload4 series. Generally the meantime values seem to not vary that much, with one exception in the OVS-cancan-1000kbit/s-payload8 experiment series.

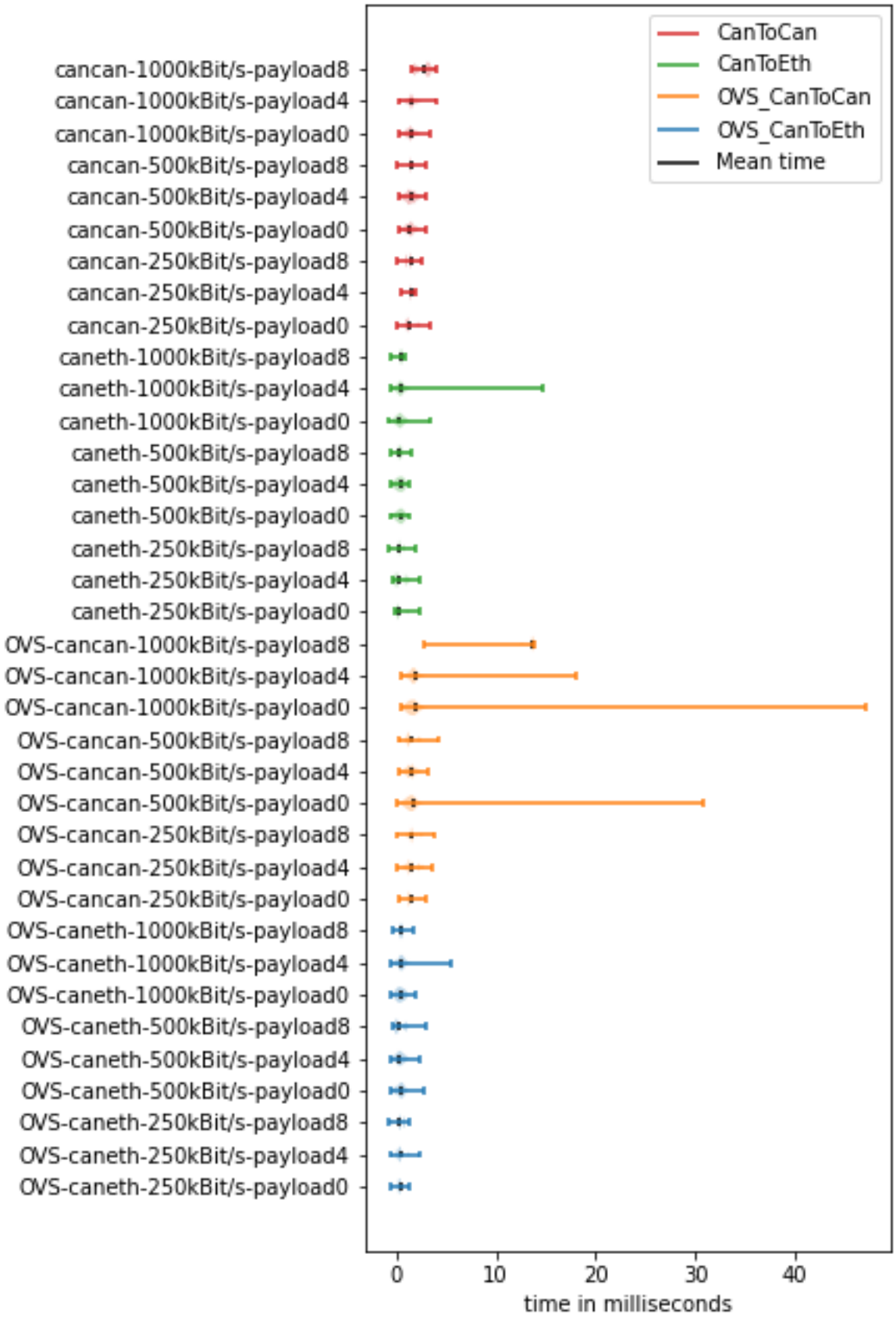


Figure 4.8: Microbenchmark on a Raspberry Pi 4.

Inspecting table 4.2.2 a possibly unexpected effect can be seen. The min values for both translator programs are always negative when sending from CAN to ethernet. Since a negative time measurement is obviously not correct an explanation is needed. Since there was no message loss the measurement needs to be wrong. Having the experimental setup in mind, the only possible reason to explain this effect is that the software used to log ethernet traffic is faster than the one used to log CAN traffic, which makes sense since optimizing ethernet drivers and tools is way more explored and therefore optimized.

This observation is also very important when analyzing the other obtained results. Having a most negative value of 0.7 milliseconds shows, that the measurement accuracy is lower than $\pm$ 0.7 milliseconds.

Another critical observation is the mean transmission time of the OVS-cancan-1000kBit/s-payload8 experimental series with a value of 13.5 milliseconds. The other translator program has a value of 2.8 milliseconds. Comparing those values the transmission time is nearly two and ten times higher than in other experimental series. Using *canbusload* in combination with *cangen* showed that the computational strong computers are able to reach a slightly higher busload when sending on maximal speed. In a real deployment the busload will never be as high as in the experiments, so this effect will not be a problem. Nevertheless, it is noticeable that the SDN optimized solution performs worse than the other.

Before starting a deeper analysis of the big maxima values or outliers seen in figure 4.8, the smaller table 4.2.2 should be watched to analyze the general behavior. That table shows the time differences between the translator program without forwarding logic and the proposed SDN optimized one. Positive values mean that the SDN optimized solution was slower and vice versa.

In all experimental series, less those having noticeable maxima, the mean time differences are lower than 0,1 milliseconds. Inspecting the maxima time differences the SDN optimized solution tendentially might be slower, but not always. Having the inaccuracies and the randomness of the scheduler of the OS in mind they perform very similar. Comparing the differences between the min values would not make sense since the inaccuracy of the measurement would be too significant.

After excluding the experimental series having noticeable maxima those still need to be analyzed.

experimental series	Δ MAX (ms)	Δ AVG (ms)
cancan/bitrate250/payload0	-0,3	0,1
cancan/bitrate250/payload4	1,7	0,1
cancan/bitrate250/payload8	1,2	0,1
cancan/bitrate500/payload0	27,8	0,2
cancan/bitrate500/payload4	0,3	0,0
cancan/bitrate500/payload8	1,2	0,0
cancan/bitrate1000/payload0	43,8	0,6
cancan/bitrate1000/payload4	14,0	0,4
cancan/bitrate1000/payload8	9,9	10,7
caneth/bitrate250/payload0	-1,1	0,1
caneth/bitrate250/payload4	-0,1	0,0
caneth/bitrate250/payload8	-0,7	0,1
caneth/bitrate500/payload0	1,5	0,0
caneth/bitrate500/payload4	0,9	0,0
caneth/bitrate500/payload8	1,6	0,0
caneth/bitrate1000/payload0	-1,3	0,0
caneth/bitrate1000/payload4	-9,2	0,0
caneth/bitrate1000/payload8	0,7	0,0

Table 4.3: Time differences between translator programs. Positive values showing slower processing using the SDN optimized gateway and vice versa.

experimental series	MAX (ms)	MIN (ms)	AVG (ms)
cancan/bitrate250/payload0	3,4	0,1	1,4
cancan/bitrate250/payload4	1,9	0,4	1,4
cancan/bitrate250/payload8	2,6	0,1	1,4
cancan/bitrate500/payload0	3,0	0,3	1,4
cancan/bitrate500/payload4	3,0	0,2	1,4
cancan/bitrate500/payload8	3,1	0,1	1,4
cancan/bitrate1000/payload0	3,3	0,3	1,4
cancan/bitrate1000/payload4	4,0	0,3	1,5
cancan/bitrate1000/payload8	4,0	1,5	2,8
caneth/bitrate250/payload0	2,4	-0,2	0,3
caneth/bitrate250/payload4	2,4	-0,4	0,3
caneth/bitrate250/payload8	1,9	-0,7	0,3
caneth/bitrate500/payload0	1,4	-0,6	0,4
caneth/bitrate500/payload4	1,4	-0,6	0,4
caneth/bitrate500/payload8	1,5	-0,6	0,3
caneth/bitrate1000/payload0	3,3	-0,7	0,4
caneth/bitrate1000/payload4	14,6	-0,7	0,5
caneth/bitrate1000/payload8	1,0	-0,6	0,4
OVScancan/bitrate250/payload0	3,0	0,3	1,4
OVScancan/bitrate250/payload4	3,7	0,1	1,6
OVScancan/bitrate250/payload8	3,8	0,1	1,6
OVScancan/bitrate500/payload0	30,8	0,1	1,6
OVScancan/bitrate500/payload4	3,3	0,2	1,4
OVScancan/bitrate500/payload8	4,3	0,3	1,5
OVScancan/bitrate1000/payload0	47,2	0,4	2,0
OVScancan/bitrate1000/payload4	18,0	0,4	1,8
OVScancan/bitrate1000/payload8	13,9	2,7	13,5
OVScaneth/bitrate250/payload0	1,3	-0,7	0,4
OVScaneth/bitrate250/payload4	2,3	-0,6	0,4
OVScaneth/bitrate250/payload8	1,3	-0,7	0,3
OVScaneth/bitrate500/payload0	2,8	-0,4	0,4
OVScaneth/bitrate500/payload4	2,3	-0,6	0,3
OVScaneth/bitrate500/payload8	3,1	-0,5	0,3
OVScaneth/bitrate1000/payload0	2,0	-0,6	0,4
OVScaneth/bitrate1000/payload4	5,5	-0,6	0,5
OVScaneth/bitrate1000/payload8	1,7	-0,5	0,5

Table 4.2: Microbenchmark on a Raspberry Pi 4 as table. Also see figure 4.8.

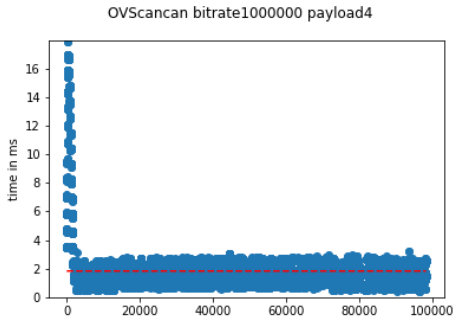


Figure 4.9: OVScancan-bitrate1000payload4

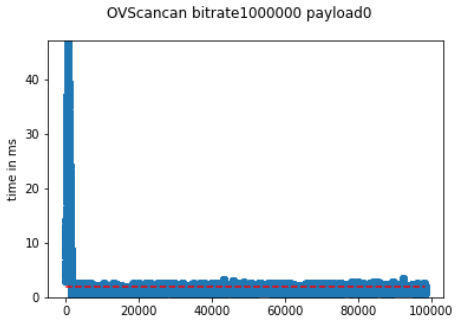


Figure 4.10: OVScancan-bitrate1000payload0

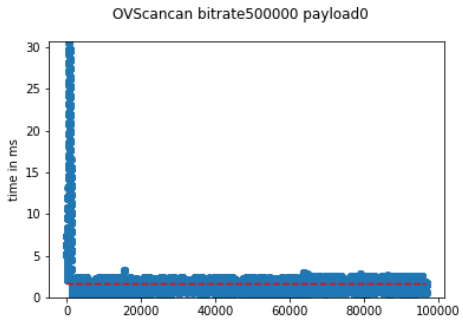


Figure 4.11: OVScancan-bitrate500payload0

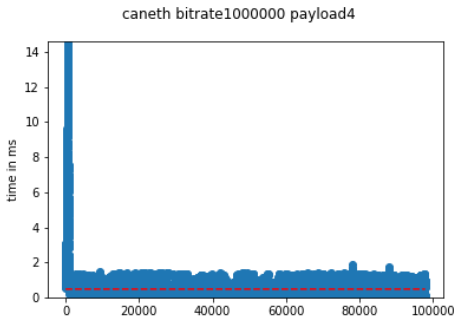


Figure 4.12: caneth-bitrate1000payload4

Till here this evaluation mostly excluded the experimental series containing big outliers. With an average transmission time of fewer than two milliseconds and outliers taking more than 40 milliseconds, they need to be discussed as well. For better understanding those experimental series are plotted showing the measured times of each frame transmitted in a chronological order showed in figures 4.9-4.12. Those figures show that all the high values were measured only at the beginning of the experiments. Two possible reasons may explain the behavior.

First, the OS is not a real-time system so possibly the scheduler needs some time to attain a good schedule and more computational intense tasks need more effort to achieve a good schedule. Since both translator programs use the same internal structure and the slow transmission at the beginning

of experiments happened more frequently in the SDN optimized solution, general computational cost optimization will very probable soften this effect.

Second, the translator programs use dynamic data structures or more precise the standard library *queue* implementation for C++. Since std::queue "acts as a wrapper to the underlying container" [4] std::dequeue, the behavior or complexity is defined by the dequeue class. The used operations random access and the insertion or removal of elements at the end or beginning have constant complexity. But the fact that the "storage of a deque is automatically expanded and contracted as needed" [3], might slow down the translator program after program start since those operations could take additional time before converging to good buffer size. Using static data structures would guarantee to never suffer from this effect.

To test those theories a new experiment will be conducted. Depending on whether the problem becomes significantly smaller or not, the theories about the cause of the problem can be confirmed or a discussion about how serious the remaining problem is can be hold. Apart from this problem, the transmission time is similar to the till now used solution and therefore, fast enough to be deployed in the experimental setup of the CoRE group.

4.2.3 Microbenchmark Gateway using Hardware 2

Purpose

Like the previous microbenchmark using hardware, this experiment will evaluate the needed transmission times over the gateway. The main focus in this experiment is on the slow transmission after starting the translater program, which is previously discussed in chapter 4.2.2. For possible reasons, scheduling issues and the use of dynamic data structures were suspected and therefore, this experiment tries to confirm those theories and determine how significant the remaining issue would be when deploying the gateway.

Procedure

This experiment is conducted in the same way as the previous one, however, a possible improvement was made in the used translator program and an additional parameter is added conducting the experiment. To improve the translator program the internally used Queue implementation from the C++ standard library was replaced by a Circular Buffer from the Boost library. To address the scheduling issues, a negative nice value is used starting the translator program.

Results

First, the previous experiment is repeated with the difference of using the improved translator program. Comparing the resulting figure 4.13 with figure 4.8 from the previous experiment, the big maxima values are still present but already significantly lower. Also, the mean transmission time for the OVS-cancan-1000kBit/s-payload8 experimental series is reduced but still over 10 milliseconds.

When plotting the measured times in chronological order as done previously in figures 4.9-4.12 it shows that the big maxima values occur again at the beginning of the experiments. So the suspected use of dynamic data structures can not directly be seen as a cause of the slow transmission after starting the translator program.

Also, the caneth-1000kBit/s-payload4 experimental series did not have big maxima this time, even though, this experimental series was obtained by conducting the previous experiment exactly as before, which shows the effect of randomness.

As the next step the improved translator program was started with a negative nice value to become prioritized by the scheduler. The resulting figure 4.14 shows that the big maxima values are not present anymore and proves the impact of scheduling algorithms.

Since the performance seems to be equal to the translator program used by the CoRE group, a replacement should be possible and will be tested as the next experiment. A more detailed analysis of the jitter is not possible with the accuracy of the measuring devices and would not make sense as long as the OS is not replaced by a real-time OS.

Typically CAN devices send traffic cyclically, as for example, the actual angle of a steering wheel in a car. Knowing how many frames are sent in a whole circle in a concrete setup, would allow to change the size of the used Circular Buffer and therefore old not needed messages would get overwritten and not sent. This possibility with a fixed schedule would perform very well for real-time communication. But with the already achieved results of never taking more than 6.6 milliseconds and on average less than 2 milliseconds, the gateway should be performant enough to be deployed in experimental setups and allow to better investigate about SDN for in-vehicle networks.

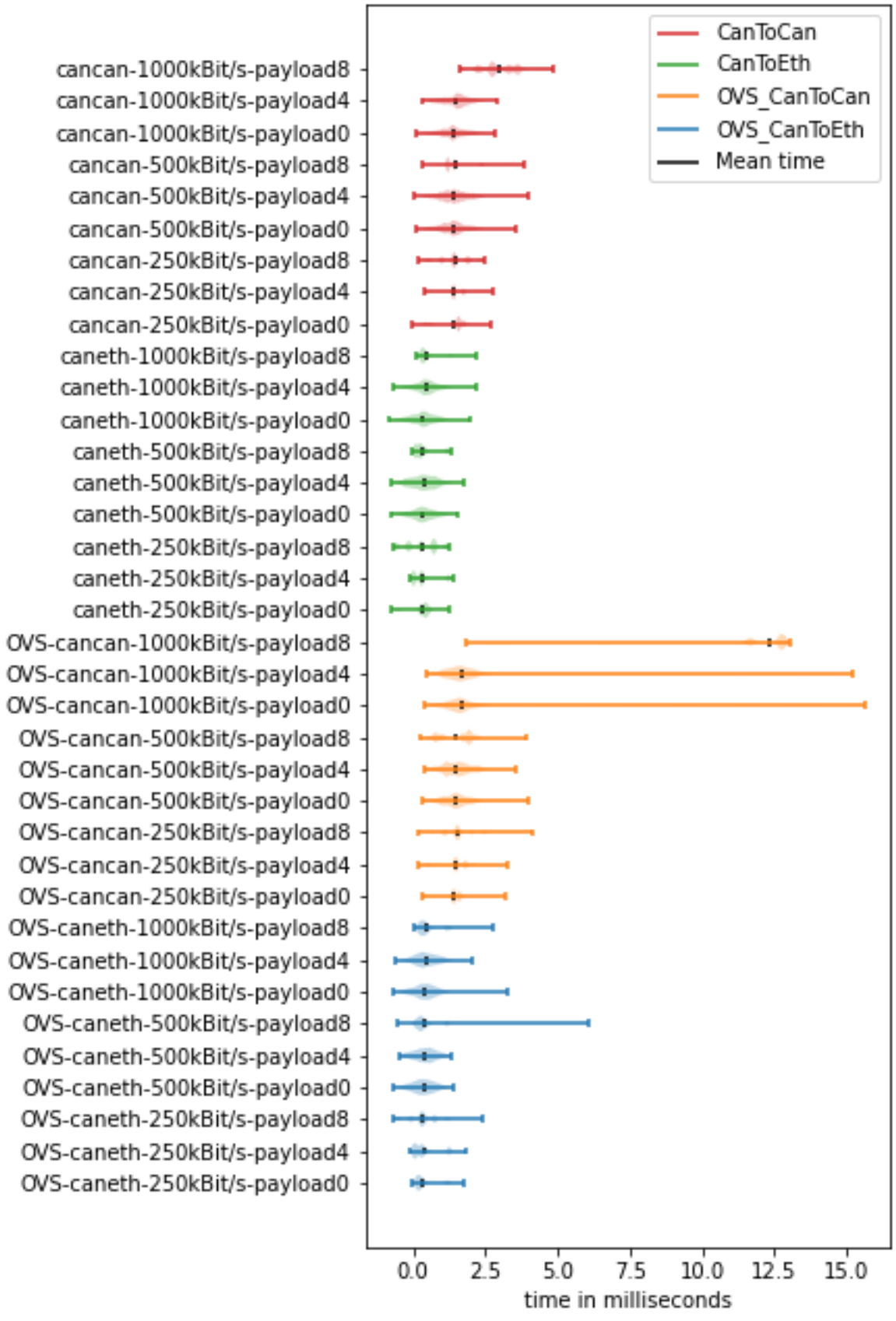


Figure 4.13: Microbenchmark 2 on a Raspberry Pi 4. No additional nice value.

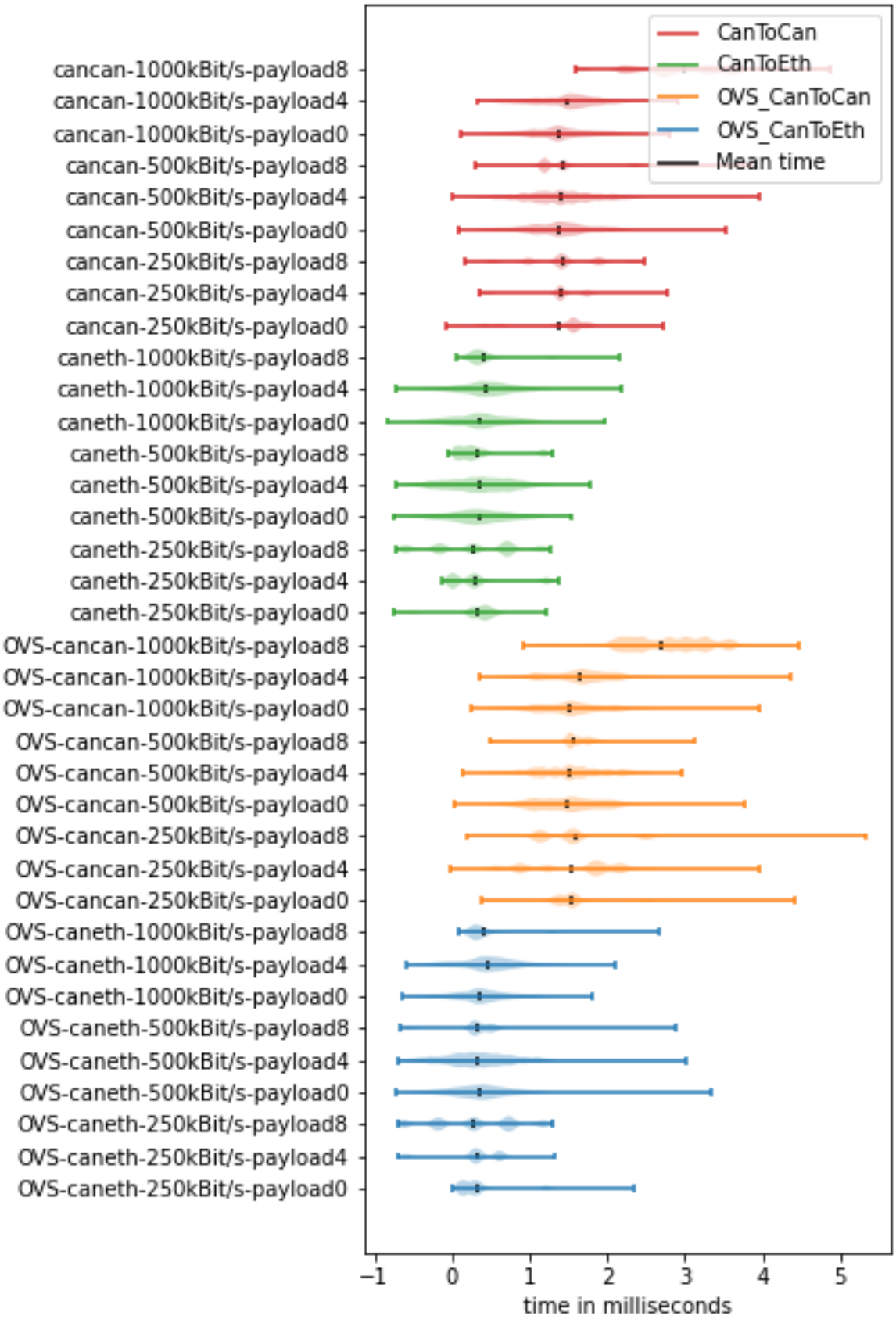


Figure 4.14: Microbenchmark 2 on a Raspberry Pi 4. Negative nice value of 20.

4.2.4 Deploying the Prototype

Purpose

The previous evaluation confirmed that the proposed gateway solution has the required performance. Since there is always a difference between experiments and real deployment, it is necessary to also deploy the gateway to confirm the functionality. Moreover, the developed prototype needs to be deployed to contribute to future research of the CoRE research group.

Experimental setup

The research group has a demonstrator car as shown in figure 4.15 and a table construction behaving as a replica, shown in figure 4.16.

Figure 4.15: The demonstrator car

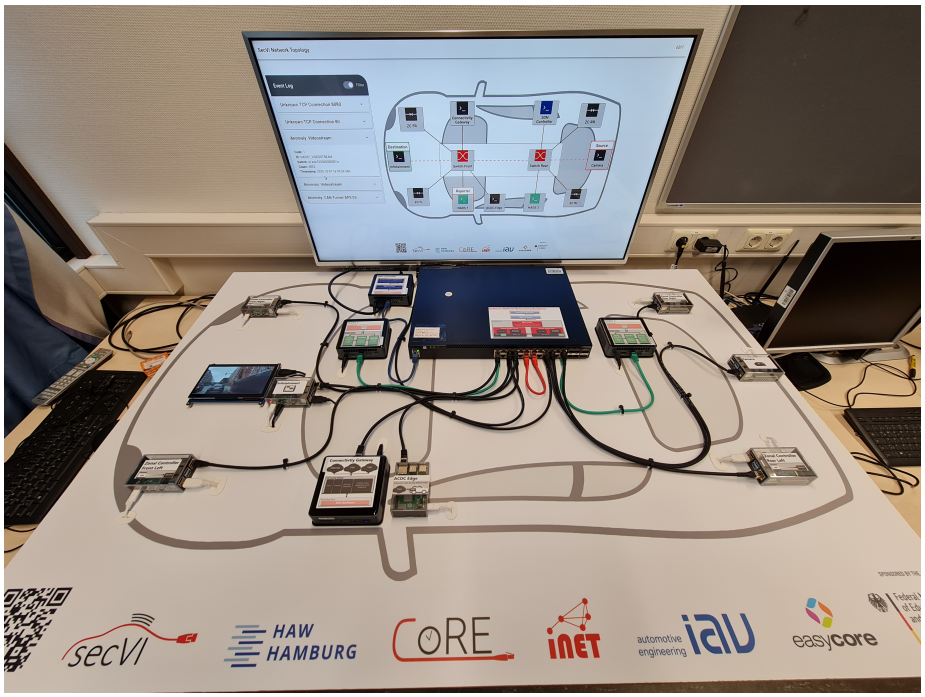


Figure 4.16: The table construction

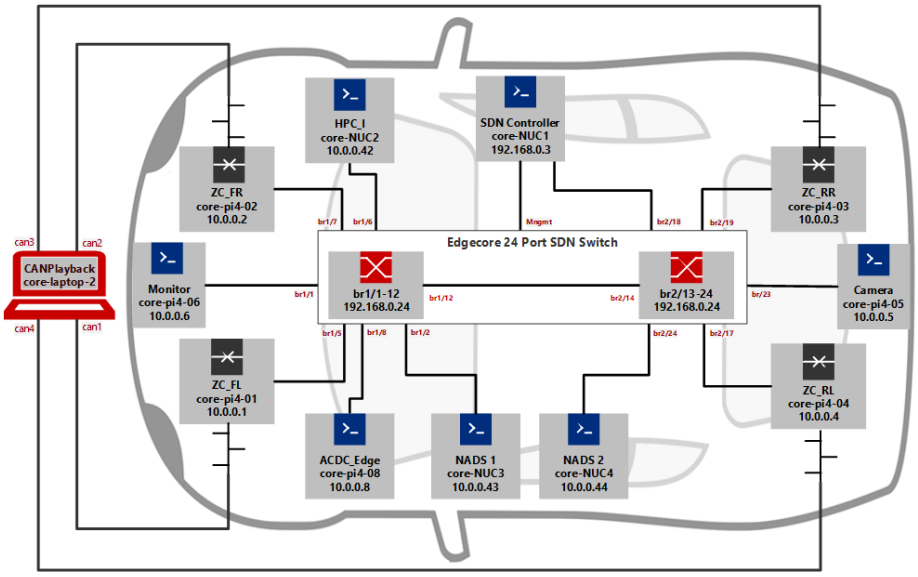


Figure 4.17: Network topology from the experimental setup

Procedure

This experiment is split into two parts. First, one gateway was completely replaced by another Raspberry Pi with the developed gateway software installed. After confirming the correct behavior, which means that the resulting traffic is equal to the solution used before, the previously used setup is reconstructed. Then the developed gateway software was installed on all gateways. The old gateway software was stopped and the new one started. When the resulting traffic over the network is equal to the traffic before replacing the gateway software, the deployment is successful.

To let the developed gateways forward the traffic according to the network policies, flow-entries representing the desired forwarding behavior needed to be created and uploaded to the SDN controller. Those flow-entries were generated with python, using the already existing flow-entries from the switch *br1/1-12* and *br2/13-24*. For a better overview refer to figure 4.17.

To be able to compare the resulting traffic when changing the gateway software, a deterministic traffic generation is needed. Therefore saved traffic files, which were created by dumping the traffic from the demonstrator car 4.15 over 15 minutes, were played. Since log files, saving 15 minutes of traffic, become very big very fast, the debug output information of the gateway software was used to monitor the amount of messages sent.

4.2.5 Result

After setting up everything correctly to replace one gateway, the result was surprising. In case of success it was expected to receive the same amount of traffic as before, or in case of a failure message loss could occur. Against expectations, more messages were sent over the network than before.

So the experiment was repeated, but saving traffic logs to analyze them with python. The expectation that a certain message possibly always got sent twice or something similar due to wrong configuration, could not be confirmed. Instead the distribution looked too random to be explained. The analysis of the log files also showed, that every message was at least sent as often as in the previously used gateway solution, which means no traffic loss.

After further experiments and reviewing configuration files the cause for having too many messages was found. The setup of the CoRE research group was containing an error. Their used local static configuration for their gateway solution was outdated and therefore, did not match the

network policies correctly. Once the configuration got updated, both gateway solutions behaved equally.

Replacing all four gateways produced exactly the same output as before, making the deployment of the developed prototype a success.

Relevant aspects of the development of the project

5.1 Beginning of the Project

In my home university, HAW Hamburg, several research groups offer interesting topics for a thesis. The group I was personally and technically interested in is called Communication over Real-Time Ethernet - research group (CoRE). I contacted the group and they offered me multiple projects to work on. All projects were related to their actual research project about the use of SDN for in-vehicle networks for future cars. To investigate they have a car from Volkswagen as demonstrator, an experimental setup as table construction reproducing the network traffic of the car, but providing better interfaces than the car does and a simulation.

My project treats the "problem" that the group already achieved to make most of the network traffic forwarding decisions via SDN, but not all. Many devices are still and will stay connected to CAN buses with the need of CAN-to-ethernet gateways. Their gateways work and the resulting traffic is already transmitted applying SDN technologies, but the decisions which CAN frames to translate and forward from the CAN bus into the SDN is done via static configuration files.

Before thinking together about possible solutions, I needed to get a deep understanding of these topics, their previous research succeeds and their actual setup. From all solutions we could think about together, I chose a low performance but very clean and safe working one. The gateway translates all CAN frames into ethernet frames and hosts a virtual switch. The switch component then interacts with the SDN via the OpenFlow protocol to choose

which frames to forward into the existing SDN. Like that the gateway itself is part of the SDN at the cost to translate many messages unnecessarily.

Having in mind that the low performance might cause that the developed gateway might not be usable in the end, the objective of this work became a usability study or analysis of the chosen gateway strategy.

5.2 Lessons Learned

5.2.1 Structural Related

As in the most projects, many mistakes could have been avoided by better analysis. Since I knew about the importance of analysis, I tried to analyze everything in detail, but I missed it in certain parts. Meanwhile I did research a lot about all related technologies and made use of software engineering strategies, I did miss to analyze the working situation. For me, it was the first time that my work participated existing work of a group, so I was also missing the experience of how important it is to analyze properly what others did already.

For that mistake, I invested a lot of time developing my first gateway approach without noticing that the existing repository which I had access to, already proposed solutions for many requirements I solved on my own. So including my gateway approach to the project repository of the CoRE group, many parts I invested a lot of time into, especially doing research, became unnecessary. Similar issues about reading official documentation about how to set up certain tools and then noticing that the Trac-WiKi already proposed guides to do what I needed without the need of deep understanding. Even though this made me use way more time to reach the same goal, I think it was beneficial for my personal learning progress and the documentation writing due to better understanding.

Another very time-consuming mistake that would have been avoidable through better analysis, was to estimate the desired format of the thesis documents. Instead of directly requesting access to the template, I just assumed a traditional paper structure and started to work with that wrong estimation. That led to many hours unnecessarily spent on research, writing and text changing.

In the project management process, I discovered an important point for the time planning. This might depend on each person, but for me, it was a mistake to estimate the needed time for tasks only in hours. I noticed that

I was not able to read scientific documents or write documentation for the same amount of time as I am generally able to work on software development. So planning to work a whole day on documenting after working several days on software developing did not accomplish the expected success. Rather than hours it worked better to measure needed time in objective-oriented working sessions, independent of how much time each of them takes. For example, to set the goal of reading two certain papers of a list of papers, instead of saying to read as many papers as possible within six hours. With that strategy, the workflow became way more efficient and the progress more predictable.

5.2.2 Technical Related

Once I decided that the final prototype will be deployed on a Raspberry Pi the choice of developing for Linux was done. Since I use Windows as OS for my personal computer I could not simply develop and test the needed translater software. As solution, I started to use a C++ IDE for Windows with the idea of compiling and running the software under Windows Linux Subsystem (WSL). I thought using an IDE under Windows and compiling it directly under the VM of WSL would be the easiest way. WSL by default does not provide a full kernel which could have been solved, but made me change to the use of a conventional VM.

Also the idea of using an IDE for Windows restricted the debug opportunities since the program could not be run inside the IDE. Additionally the IDE comfort worsened due to error messages caused by not found library calls etc. Probably it would have been possible to solve that issue by installing a cross compiler but this did not appear to be the easiest solution. Considered as the probably simplest solution, I installed an IDE in the VM. Developing in an IDE inside a VM, meanwhile all other programs were used in the host OS felt uncomfortable, so I installed an X11-Server tool on the host OS Windows. Like that, I was able to use the IDE in the VM having the GUI of the IDE opened, as if it would be any other program of the host OS.

Even after such a long way of searching for the optimal workflow, later on, I had to change the whole setup again. The VM together with the SDN controller service needed more RAM than my personal computer could provide to work smoothly. So after that many tries around, I just removed everything and installed a full Linux OS as dual-boot. Apart of resizing the partition, this setup could be used till the end of the project.

Generally, I am still convinced that thinking a lot about finding the optimal working flow before starting the work is a good idea. As the reason for the inefficiency and the amount of withdrawn decisions, I suspect missing experience and knowledge. Due to all things learned in that process in future projects I would clearly do better.

Another big issue occurred while conducting the second experiment 4.2.2. At that point I used two computers to conduct the experiment. One computer sent frames to a CAN bus and also read their arrival time and their arrival time on the destination CAN bus. The other one translated and forwarded them. The measured times were growing infinitely independent of which translator implementation or CAN bus configuration I tried. This was caused by the unexpected behavior of the Linux CAN driver implementation. When sending and reading from one CAN interface with the same device, the CAN frames get readable earlier than they are sent. Once finally being aware of that effect, a third computer was used to generate the traffic and the experiment generated usable results.

Conclusion and Outlook

The use of real-time ethernet for in-vehicle backbone networks is the subject of much research. For the gateways, which are necessary to transfer data over those backbone networks, several approaches are existing and investigated. This work contributes to the research of using a Software Defined Network as an ethernet backbone network and investigates the approach of making the gateway part of the SDN, instead of programming the gateway statically.

As part of this work a strategy to include a CAN ethernet gateway into an SDN was elaborated. Additionally, a protocol translation strategy and the according software were implemented. Several evaluations were performed confirming the potential of the approach. Finally, a prototype was implemented in hardware and tested in an experimental setup.

The results of this work make it possible to program CAN-ethernet gateways via standardized SDN-related protocols, such as OpenFlow, in experimental setups. Here the use of this gateway strategy grants the gateways all the advantages of SDN, without any disadvantages.

The elaborated strategy to include the gateway into the SDN consists of translating every CAN frame into ethernet frames and forwarding them to a virtual switch. Since this work showed that the resulting computational overhead did not lead to poorer performance in the environments tested, it is expected that the strategy presented will be used for further research.

Although this work is limited to CAN-Ethernet, the presented approach can also be implemented for LIN and many other protocols. This increases the potential of this strategy even further, since all the different domains and protocols can be controlled via the same interface of the same SDN controller. Having a central interface to program all components of the

network while benefiting from features like fail-state operations will improve future research.

Bibliography

[1] Git Community. git –everything is local. www.https://git-scm.com.

[2] Steve Corrigan. Introduction to the controller area network (can). 2002.

[3] cppreference.com. std::dequeue - cppreference.com. https://en.cppreference.com/w/cpp/container/dequeue.

[4] cppreference.com. std::queue - cppreference.com. https://en.cppreference.com/w/cpp/container/queue.

[5] Armando Astarloa Cuéllar. Development solution for next generation ethernet using tsn.

[6] Open Networking Foundation. Open network operating system (onos) sdn controller for sdn/nfv solutions. https://opennetworking.org/onos/.

[7] Peter Fussey and George Parisis. Poster: An in-vehicle software defined network architecture for connected and automated vehicles. In *Proceedings of the 2nd ACM International Workshop on Smart, Autonomous, and Connected Vehicular Systems and Services*, CarSys '17, page 73–74, New York, NY, USA, 2017. Association for Computing Machinery.

[8] CoRE Research Group. Communication over real-time ethernet group. https://core.informatik.haw-hamburg.de.

[9] Marco Haeberle, Florian Heimgaertner, Hans Loehr, Naresh Nayak, Dennis Grewe, Sebastian Schildt, and Michael Menth. Softwarization of automotive e/e architectures: A software-defined networking approach. In *2020 IEEE Vehicular Networking Conference (VNC)*, pages 1–8, 2020.

[10] International Organization for Standardization. *Road vehicles — Interchange of digital information — Controller area network (CAN) for high-speed communication*. International Organization for Standardization, iso 11898-1:1993 edition, 1993.

[11] International Organization for Standardization. *Road vehicles — Interchange of digital information — Controller area network (CAN) for high-speed communication — Amendment 1*. International Organization for Standardization, iso 11898:1993/amd 1:1995 edition, 1995.

[12] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[13] Eleftherios Kyriakakis, Maja Lund, Luca Pezzarossa, Jens Sparsø, and Martin Schoeberl. A time-predictable open-source ttethernet end-system. *Journal of Systems Architecture*, 108:101744, 2020.

[14] Marco Di Natale, Haibo Zeng, Paolo Giusto, and Arkadeb Ghosal. *Understanding and Using the Controller Area Network Communication Protocol*. 2012.

[15] Suk-Hyun Seo, Jin-Ho Kim, Sung-Ho Hwang, Key Ho Kwon, and Jae Wook Jeon. A reliable gateway for in-vehicle networks based on lin, can, and flexray. *ACM Trans. Embed. Comput. Syst.*, 11(1), April 2012.

[16] trac. Welcome to the trac open source project. https://trac.edgewall.org/demo-1.4.

[17] Open vSwitch Community. Open vswitch. https://www.openvswitch.org/.

[18] Samuel Woo, Hyo Jin Jo, and Dong Hoon Lee. A practical wireless attack on the connected car and security protocol for in-vehicle can. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):993–1006, 2015.

[19] Weiying Zeng, Mohammed A. S. Khalid, and Sazzadur Chowdhury. In-vehicle networks outlook: Achievements and challenges. *IEEE Communications Surveys Tutorials*, 18(3):1552–1571, 2016.

[20] Lin Zhao, Feng He, Ershuai Li, and Jun Lu. Comparison of time sensitive networking (tsn) and ttethernet. In *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, pages 1–7, 2018.

UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

TFG del Grado en Ingeniería Informática

Developing a CAN Ethernet Gateway for Software Defined Networks in Future Cars Documentación Técnica

Presented by Yunus Ülker
in the Universidad de Burgos — May 25, 2022
Tutor: Juan Jose Rodriguez

Contents

List of Figures

List of Tables

Appendix A

A Software Project Plan

A.1 Introduction

This section mainly focuses on explaining the project management and the time planning. The used work breakdown-structure for time management is shown in its initial and final state. Also the resulting costs, including a legality analysis, for the project are shown and explained.

A.2 Project Management

For all major projects good project management is essential to success. IT project management is typically broken down into five related stages which are Initiation, Planning, Execution, Monitor and control, and Closing. The initiation phase corresponds with the topic and supervisor finding. Planning is certainly the most important part of the project management process and is the base for all other stages.

The key to good planning is taking into account as many variables as possible. Traditional IT project management often suffered from the mistake of planning too static too far to the future. This is why agile project management frameworks, e.g Scrum, were gaining more popularity in the last years. Simply using Scrum would not work since in this particular project there is no team but only one participant. A specialty about this project is that it is run as part of a research group offering guidance, support, and a place for discussions. For the most projects cost planning or resource and time planning is the main limiting variable. This limitation usually comes with a fixed goal which needs to be reached at the end of the project.

This project does not have a limited budget of hours that can maximally be spent to finish the project, but two possible deadlines. The first would have been January 2022 and the second June 2022. Consulting the estimation of the research group leading professor, finishing the project by January would have been a challenging goal.

Inspired by agile project management concepts and by making use of not being dependent on a team, I chose to plan the project adaptable to any changes and tried to not specify deadlines for specific tasks too detailed. So an ordered list of milestones without fixed dates was used to determine the progress of the project. To stay agile, those general milestones were set to be broken down into multiple concrete goals later on. In that way new information learned in the progress could be directly implemented in the project and avoid wrong estimations.

To reach those milestones and keep track of the project's progress, I split my working time into two different repeating phases, preparing and working. In the preparing phase I analyzed the recent progress and any remaining tasks. By breaking down those remaining tasks and possibly adding new ones, I created a list of goals to complete before the next preparation phase. Even though this technique is inspired by sprints from the Scrum framework, it is different since assigning a certain amount of time is an essential component of a Scrum sprint. When working without a team I did not see any advantage of assigning time to a sprint duration but a loss of flexibility and unneeded overhead.

Even if this work has only one participant many meetings were held with other members of the research group. Very helpful for this project was the way of communication inside the group. Nearly every day the two group leading Ph.D. students were available in a Teams room for several hours and every member was invited to join whenever they want.

Using those schemes till the end of the project makes the closing-stage trivial since its objectives can be treated equally as any other before.

A.3 Time Management

To monitor the progress of the project a work breakdown structure (WBS) was used. As explained in the previous section the project management was agile and therefore the initial WBS was very empty and is shown in figure A.1. The WBS grew in the progress of the project and the final state is shown in figure A.2 and gives an overview of the time management of the whole project.

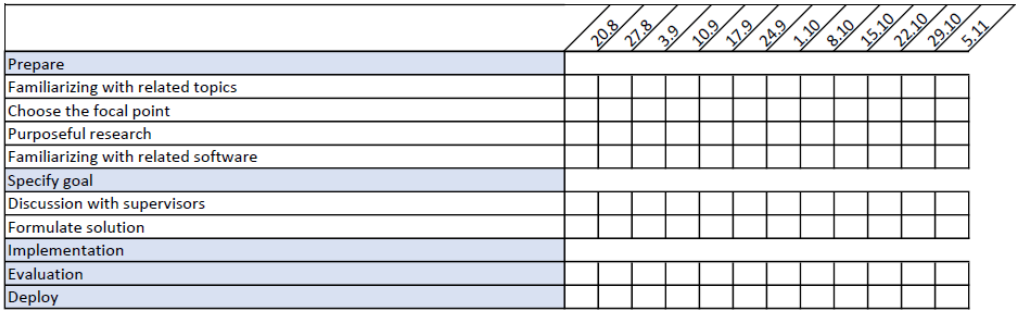


Figure A.1: Initial WBS

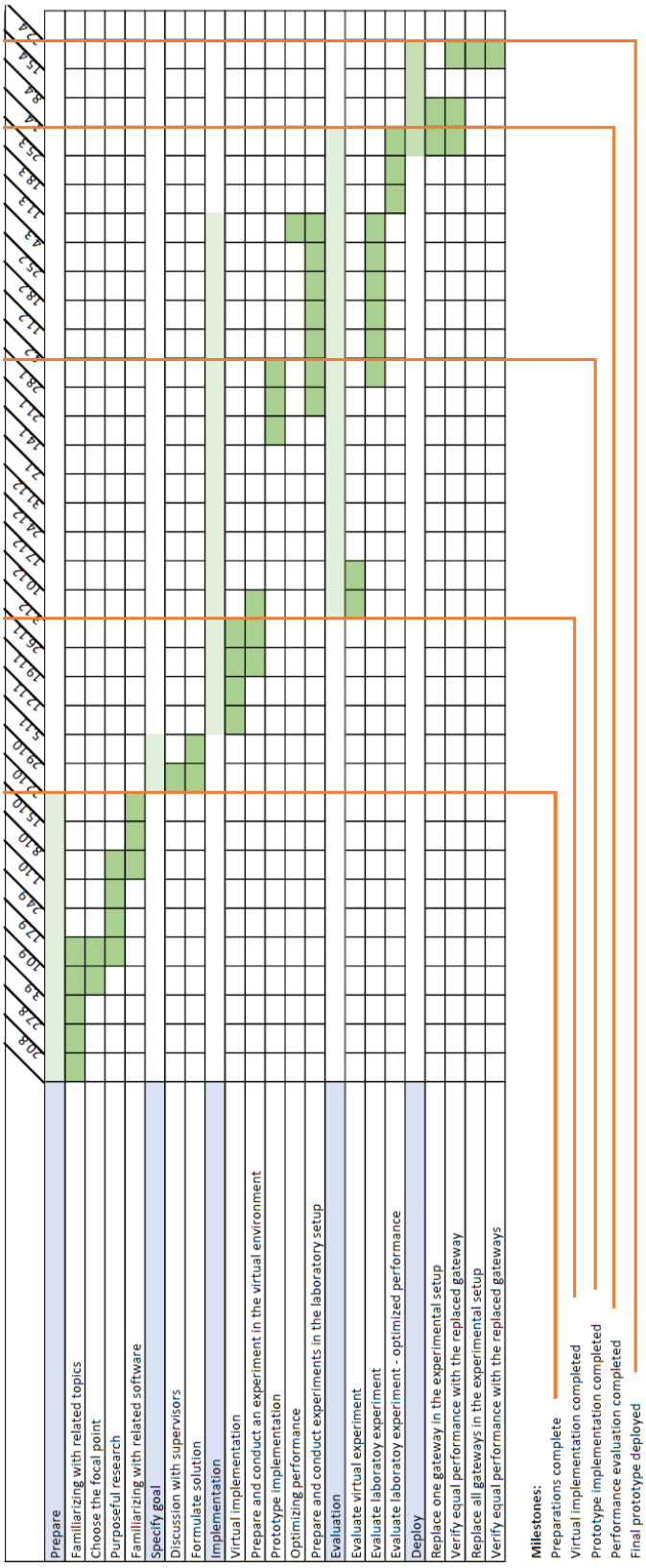


Figure A.2: Final WBS

A.4 Feasibility Study

A.4.1 Costs

To discuss the costs of this project they are split into two categories, personnel and hard- and software costs. To estimate personnel costs the ECTS points can be converted into hours to be paid and a salary between 14 and 20 Euro per hour can be estimated. This calculation formula leads to 5040-7200 € excluding possible taxes which would depend on other factors like the type of contract and additional income of the employee.

The experiments were conducted using different hard- and software setups resulting in different costs. Those are shown separately in cost-tables having the same names as the corresponding evaluation in the Memoria document.

All those listed prices refer to the used components and not the cheapest possible solution to do it. Especially the listed *Lab Laptop*[12] and *Personal Laptop*[13] could be replaced by cheaper computers. Also replacing the *6xCAN-1xUSB Adapter*[9] with two *CAN-USB-Adapter*[11] would make the setup *Microbenchmark Gateway using Hardware* A.4.1 cheaper. Those factors might be interesting to copy the experiment, but in the case of this work it was irrelevant since all related software and hardware costs were paid before and independent of this project.

Description	Price per Unit	VAT	Quantity	Price
Personal Laptop [13]	667,38 €	126,80 €	1	794,18 €
Total cost				794,18 €

Table A.1: cost table: Micobenchmark Gateway virtual

Description	Price per Unit	VAT	Quantity	Price
CAN-USB-Adapter [11]	180,00 €	34,20 €	3	642,60 €
6xCAN-1xUSB Adapter [9]	735,00 €	139,65 €	1	874,65 €
Raspberry Pi[10]	84,50 €	16,06 €	1	100,56 €
Lab Laptop [12]	2.161,96 €	410,77 €	1	2.572,73 €
Personal Laptop [13]	667,38 €	126,80 €	1	794,18 €
Total cost				4.984,72 €

Table A.2: cost table: Micobenchmark Gateway using Hardware

Description	Price per Unit	VAT	Quantity	Price
SDN switch [1]	954,00 €	181,26 €	1	1.135,26 €
Switch OS [2]	642,99 €	122,17 €	1	765,16 €
Support service for switch OS[3]	130,21 €	24,74 €	1	154,95 €
NUC mini computer kit [4]	234,45 €	44,55 €	4	1.116,00 €
SSD for the NUC [5]	23,52 €	4,47 €	4	111,96 €
RAM for the NUC[6]	26,46 €	5,03 €	4	125,96 €
Network cable [7]	5,00 €	0,95 €	10	59,50 €
Power supply cables [8]	50 €	950 €	1	59,50 €
6xCAN-1xUSB Adapter [9]	735,00 €	139,65 €	1	874,65 €
Raspberry Pi [10]	84,50 €	16,06 €	6	603,33 €
CAN-USB-Adapter [11]	180,00 €	34,20 €	4	856,89 €
Lab Laptop [12]	2.161,96 €	410,77 €	1	2.572,73 €
Personal Laptop [13]	667,38 €	126,80 €	1	794,18 €
Total cost				9.229,98 €

Table A.3: cost table: Final Prototype Deploy

A.4.2 Legal Viability

The developed gateway is not and will not be made public. All the used hard- and software were accessed legally. This work only contributes to research.

Also it is expected that if companies would use this gateway strategy in the future, they would implement their own specific hardware with software developed specifically for their devices.

[1] Edgecore 24x 1GbE RJ45 SDN Switch (4610-30T-O-AC-F)
[2] Pica8 PicOS 1GE 24-Port Switch Enterprise Edition Network OS (P-OS-1G-EE-24)
[3] Support for P-OS-1G-EE-24 (P-OS-1G-EE-24-S1)
[4] Intel® NUC Kit NUC8i3BEK2, Barebone (NUC8i3BEK2)
[5] SSD for Intel NUC (HP EX900 120 GB M.2 2280, PCIe 3.0 x4)
[6] Crucial SO-DIMM 8 GB DDR4-2400
[7] RJ45 Patch cabel
[8] Power supply multi socket for 1x Switch, 4x NUC, 6x PI
[9] IPEH-004062 PCAN-USB X6 CAN-FD D-SUB
[10] Raspberry PI 4 B 4 GB All-In-Bundle (RPI 4B 4GB ALLIN)
[11] IPEH-002021 PCAN-USB-Adapter
[12] ThinkPad P1 P1000
[13] Asus Zenbook 14 UX430UA

Appendix B

Software Requirement Specification

B.1 Introduction

This section describes the general goal and which requirements to meet. Those requirements are divided into functional and non-functional and are listed separately. In addition to the list of requirements, several requirements are explained in more detail.

B.2 General Objectives

The general goal is the research, development, and evaluation of a CAN ethernet gateway with a specialization in the use of SDN technology. More specifically, the forwarding logic is not intended to be part of the gateway itself and instead becomes part of an SDN controller, as shown in figure B.1.

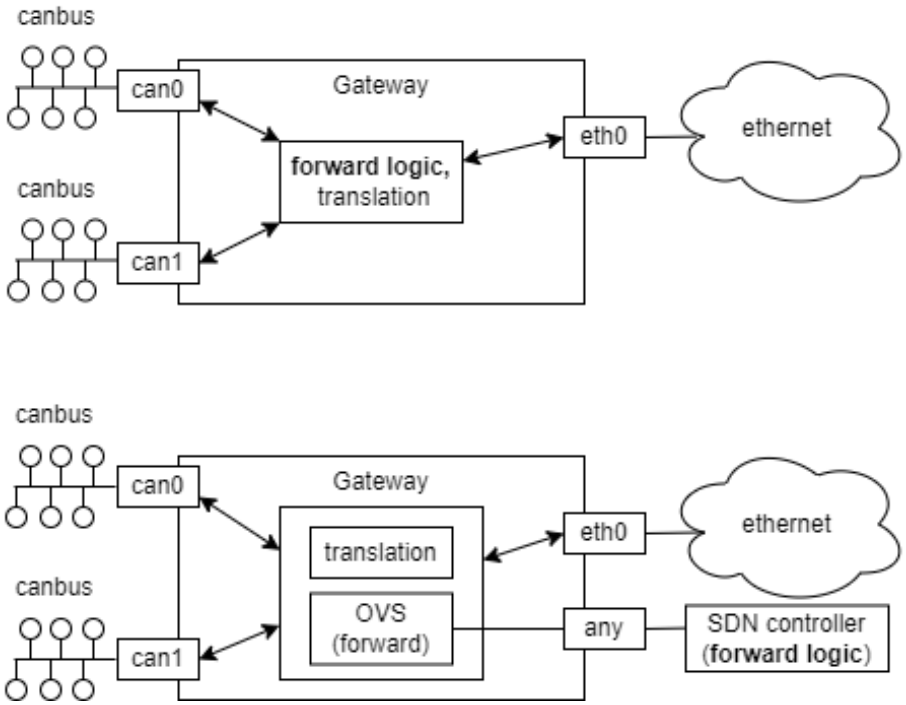


Figure B.1: Basic idea of this gateway strategy.

If these SDN optimized gateways perform well enough, they will replace the gateways used in the experimental setup used by the CoRE research group, allowing them to continue the research of this work. Developing a gateway with the aim of deploying it in a car would lead to a huge amount of requirements, but developing a prototype for research purposes does need not fulfil most of them. However analyzing characteristics, behavior and potential upgrades becomes a requirement instead.

B.3 Software Requirement Catalogue

B.3.1 Non-functional Requirements

- No packet loss in the gateway.
- Similar latency to the existing solution used by the CoRE group.
- To allow the gateway to connect CAN buses with an ethernet network, those networks are needed first of all.

- Also, the gateway needs to be connected to both networks.
- Since CAN ports are rare to find in computers, additional CAN to USB adapters might be necessary.
- The gateway forwarding behavior is defined by an SDN controller, which itself is not part of the gateway. So an SDN controller needs to be reachable via the connected ethernet network.
- Having terminal access with root rights on the gateway is necessary.

B.3.2 Functional Requirements

- Change forwarding rules
  What is special about this gateway is the possibility to change the forwarding rules at runtime via the interfaces provided by the chosen SDN controller.
- Protocol translation
  Sending CAN frames using the ethernet protocol requires a bidirectional translation strategy.
- Forwarding traffic
  When receiving traffic the gateway needs to act according to the forwarding rules. Depending on the receiver and sender interface *Protocol translation* might have to be applied.
- Multicast forwarding needs to be supported.

B.4 Requirement Specification

B.4.1 Setting up a Gateway

Terminal access with root rights is required. The gateway needs to be connected physically to the ethernet network and minimum one CAN bus. According to the individual network configuration, the ethernet port might need to be configured with a certain IP to be able to communicate with the SDN controller. For the connection to the CAN bus probably additional hardware and possibly additional drivers need to be installed. Also the CAN interface needs to be configured with the same bitrate as the other devices connected to that bus.

For each connected CAN bus it is necessary to create virtual ethernet pair links, with one port to be used from the virtual switch and one from the translator program. To do so it is required to install and use the Open vSwitch software to create a virtual switch in the gateway. How to create virtual ethernet pair links and how to create and configure a virtual switch using Open vSwitch is briefly described in chapter D.3.4 and chapter D.4.

The translator program needs to be compiled once, and executed for each connected CAN bus with parameters according to the network configuration, including the custom source MAC that the gateway should use when sending traffic over ethernet, and which ports to use for CAN and ethernet.

B.4.2 Change Forwarding Rules

The use of an OpenFlow supporting virtual switch for forwarding decisions makes the forwarding behavior of the gateway act accordingly to the OpenFlow standard. When choosing ONOS as the SDN controller, as in the experimental setup used for this work, there are three different interfaces available to communicate with the controller, which are a command-line interface (CLI), a REST API, and self-developed network applications with the possibility to provide a graphical user interface.

B.4.3 Protocol Translation

Ethernet frames have a field called ethertype exposing the encapsulated protocol. This field is set to CA01 to mark that the frames are representing CAN frames. How to write or read CA01 typed frames is explained in Technical Programming Documentation D.3.1.

B.4.4 Forwarding Traffic

The obvious problem is that in the context of CAN there are no SDN solutions available. To still make use of SDN each CAN interface is mirrored on a virtual ethernet interface. For those ethernet interfaces SDN solutions can be applied.

To do so the translating instances translate any CAN frame and send them to their connected virtual ethernet interface and the other way around. These frames arrive at the virtual switch as any other ethernet frame would, and the switch is able to forward them according to its configuration. Dropping those frames or sending them over the ethernet port into the in-vehicle network is trivial. If the switch instead forwards them to a virtual

ethernet interface, the belonging translator instance will translate them back into CAN frames and send them on their according CAN interface.

This strategy makes the gateway work completely according to SDN policies for CAN and ethernet traffic.

Appendix C

Design Specification

C.1 Introduction

This section briefly describes how the gateway works. Therefore possible forms of data presentations are mentioned. The procedures and the architecture are explained using figures. More details are provided in section D. A basic knowledge about Software Defined Networking and the OpenFlow protocol are assumed.

C.2 Data model

One gateway consists of one virtual switch, translating units, and network interfaces. As a virtual switch, Open vSwitch is used. In the scope of this work only the switches user interfaces are used, without knowledge about how the data is represented internally. The network interfaces need to be created according to the individual network configuration and the translating units only have a few starting parameters which depend on the individual network configuration. In most cases forwarding rules take the biggest amount of data. In this work, they are represented by OpenFlow flow-entries formatted as JSON or generated via ONOS applications.

For the network configuration any form of documentation can be used. The experimental setup of the in-vehicle network is documented using TRAC on a private website, meanwhile for small experimental setups a simple textfile was used.

C.3 Procedural Design

Before a gateway can be used its architecture needs to be set up. Therefore one Open vSwitch instance needs to be started. For each connected CAN bus one virtual ethernet link, accessible via a pair of virtual interfaces, needs to be created. Those links are used to connect translator instances and the virtual switch. The switch needs to be configured to be able to communicate with the SDN controller of the network. A translating instance needs to be started for each CAN bus.

Figure C.1 visualizes the message flow of an incoming CAN frame on a gateway with two CAN buses connected. The used topology to is shown and explained in the next section C.4.

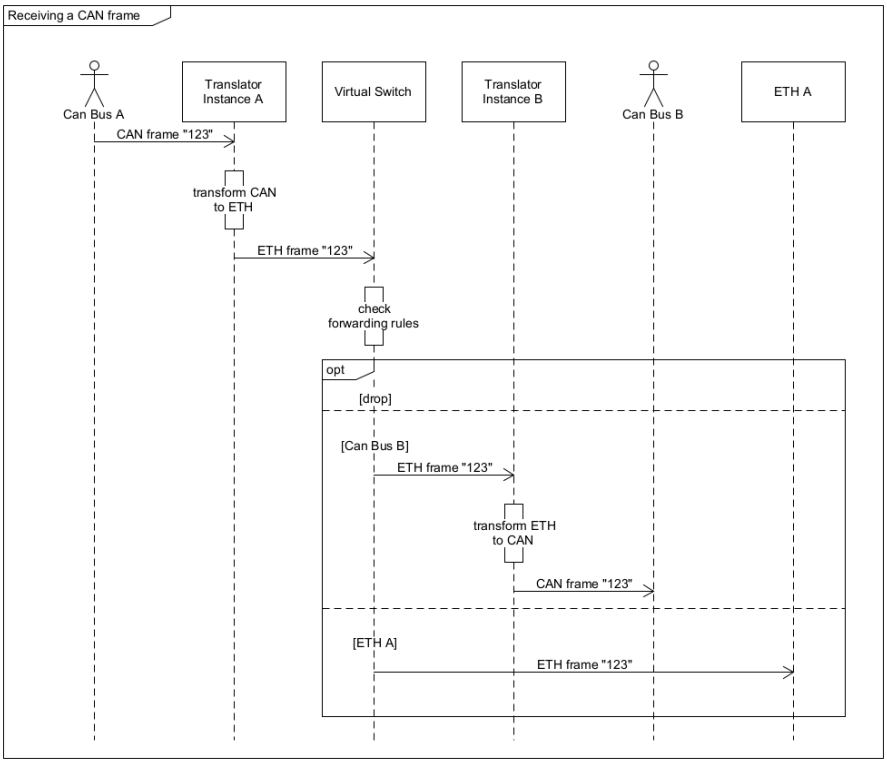


Figure C.1: UML Sequence Diagram, Receiving a CAN Frame

Once everything is set up every CAN frame received on a CAN bus will be translated into an ethernet frame and will be sent to the CAN bus according virtual ethernet link. The switch will receive those and check which are meant to be forwarded or not and do so. If the switch forwards

frames to a virtual ethernet link the corresponding translating instance will receive, translate and forward the traffic to the CAN bus. Dropping or forwarding frames over a nonvirtual ethernet interface behaves as usual.

C.4 Architectural Design

The amount of CAN buses connected to a gateway determines its architecture. The choice of how many CAN buses to connect depends mainly on the personal network setup, but also the availability of physical interfaces. The architecture of a gateway unit with two CAN buses connected is demonstrated in figure C.2.

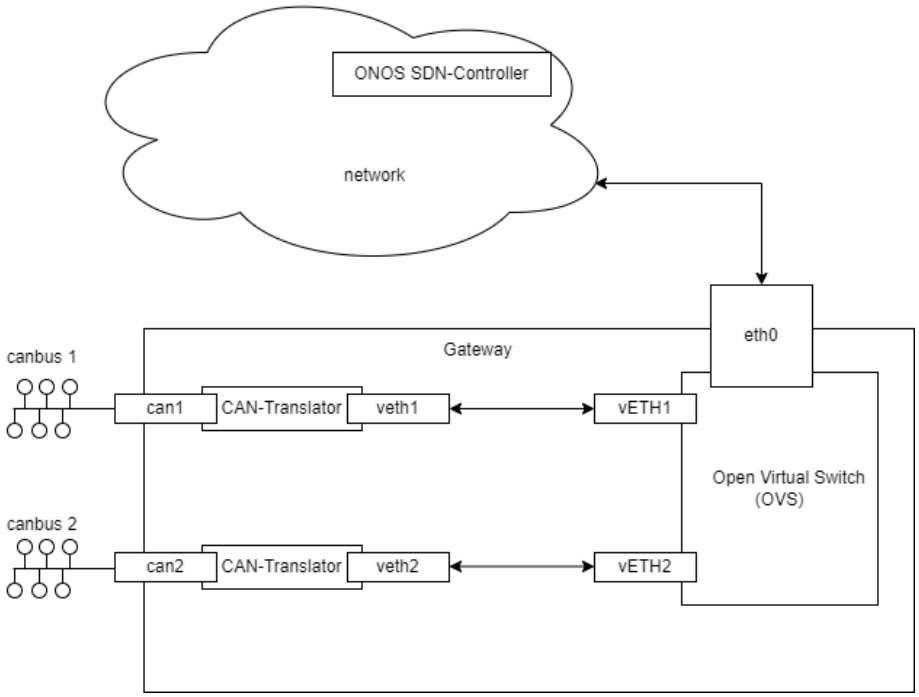


Figure C.2: Example Gateway Architecture.

For each CAN bus connected, a translator instance needs to be started. To allow the translator instances to communicate with the virtual switch a virtual ethernet link needs to be created. One virtual ethernet link consists of two virtual interfaces. One needs to be used by the virtual switch and one by a translator instance.

The translator units and their internal architecture are explained in section D.3.2. Summarized the translator program interprets the command

line arguments to know which interfaces to use and to obtain some other information. Then six threads are started with three for each direction of translating. One thread to send, one to receive, and one for the protocol translation itself.

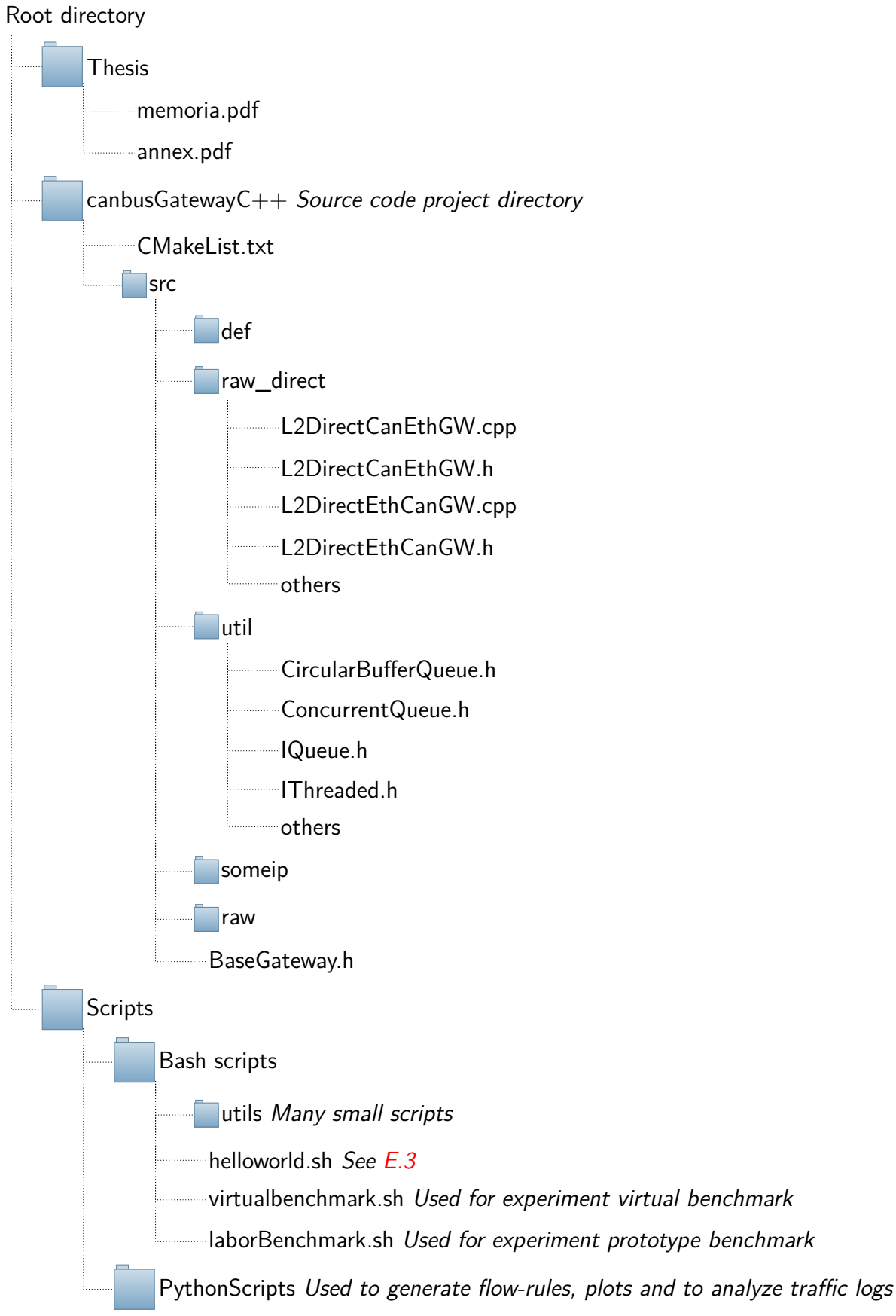
Appendix D

Technical Programming Documentation

D.1 Introduction

In this section the submitted files within their directory structure are briefly described. The protocol translation is explained in detail. Also, an explanation to compile and start the developed gateway software is proposed with an additional UML class diagram and an introduction for a fast understanding of the source code. Basic introductions on how to set up the virtual switch software and possibly an SDN controller are provided.

D.2 Directory Structure



D.3 Programmer's Manual

D.3.1 Protocol Translation Strategy, Ethertype ECA1

As mentioned in the theoretical concepts chapter *Controller Area Network* in the Memoria document, CAN supports extended and basic frames. Figure D.1 shows the different fields of an ethernet 802.1Q, a CAN basic and a CAN extended frame. The color scheme, that is used to highlight fields, is helpful to understand which fields are mapped into which.

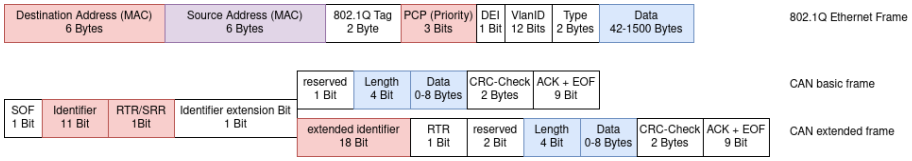


Figure D.1: Shows the Mapping of CAN and Ethernet Frame Fields

The destination MAC field is used to represent the *CAN identifier*. The identifier is stored in the last four octets of the destination MAC. The MSB of those four octets is used as a flag to show if this ethernet frame represents an extended or a basic frame. So that MSB corresponds with the SRR bit of a CAN frame. The first octet of the destination MAC is always set to 0xFF and the second to 0x00. Setting the two least significant bits in the first octet marks the frame as a multicast frame and shows that the MAC address is self-chosen. Apart from those two bits the chosen 0xFF00 for the first two octets of the destination MAC were freely chosen.

The source MAC field is used to represent on which CAN bus the frame was received originally. The source MAC field can be freely chosen, but needs to accord to the network policies. It is recommended to set the second least significant bits in the first octet, to show that it is a self-chosen MAC.

The ethertype is set to 0x8100 which means that this frame is a 802.1Q frame. The priority field is the identifier divided by 256 with a maximum of 7. This formula to set priorities may be changed in the future. The 802.1Q-ethertype field is set to 0xECA1 to mark that the frame accords with the translation strategy explained here.

In the data field the first three bytes are 0. The fourth byte corresponds to the length field of a CAN frame. Depending on that value 0-8 bytes follow representing the data field of the CAN frame.

An example of a CAN frame translated as ethernet frame is shown if figure D.2.

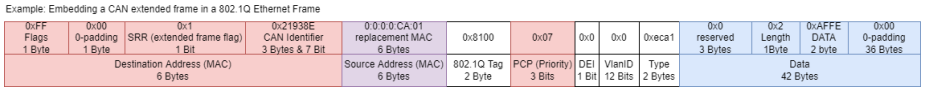


Figure D.2: Example: A translated CAN Extended Frame as 802.1Q Ethernet Frame.

D.3.2 Translator Program

For version control the private git repository of the CoRE research group is used. Since the software is part of the research group and is not meant to be published, only a snapshot of the project is part of the submission. The project is developed in C++ and can easily be imported due to the provided CMake project file.

The main method as the entry point of the translator program reads and parses command-line arguments. Then it creates two *IThreaded* objects. In this project plenty classes implement this interface, but to use the translating strategy proposed by this work *L2DirectCanEthernetGW* and *L2DirectEthernetCanGW* objects are created. Which two IThreaded objects to create depends on the command line argument *-m –mapping*. Since the different mapping strategies have a lot in common they inherit from an abstract class *BaseGateway* which implements the IThreaded interface.

All important classes and their relations can be seen in figure D.3.

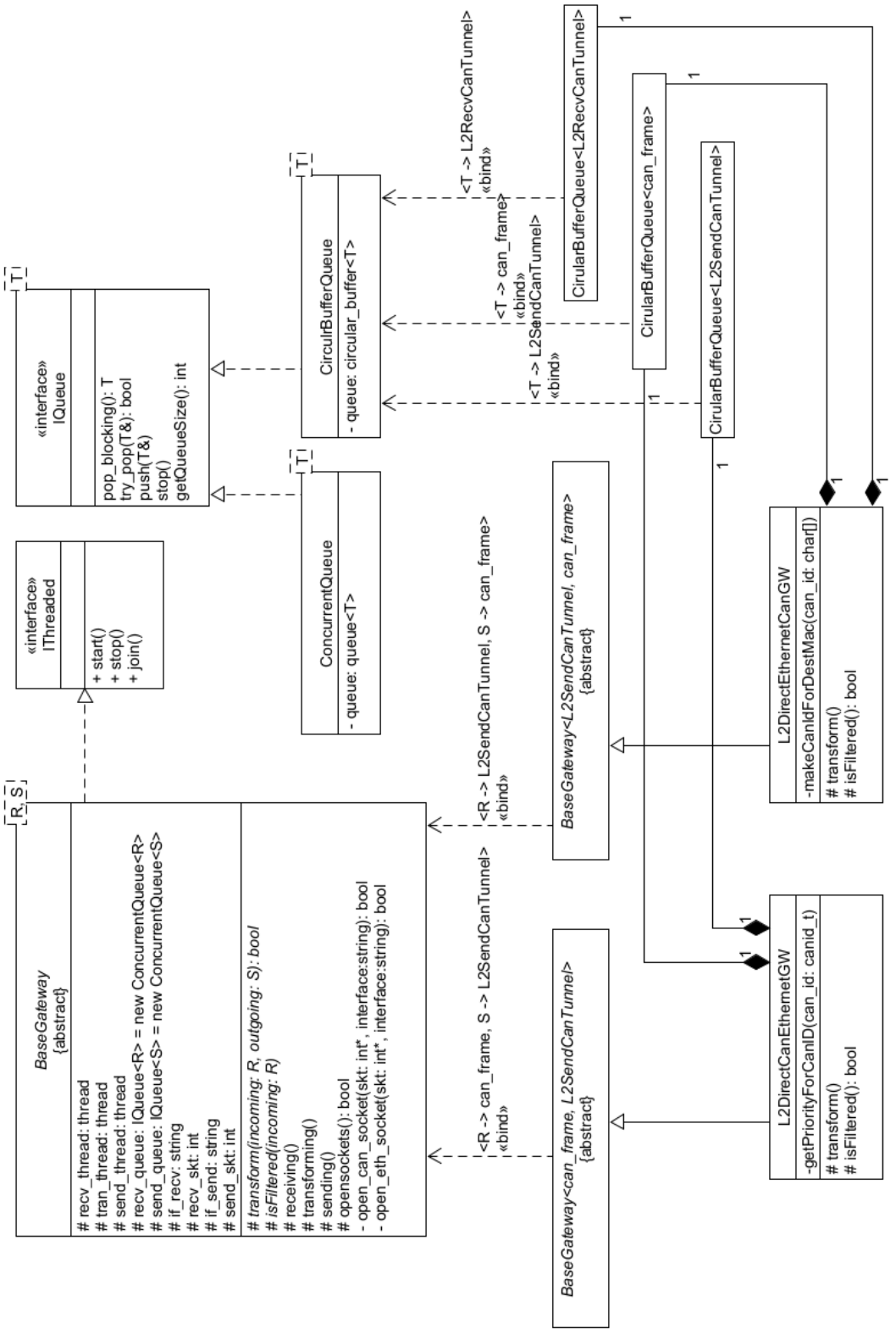


Figure D.3: Simplified UML Class Diagram from the Translator Program

The IThreaded interface provides the thread typical methods start, stop, and join. Upon starting, the BaseGateway opens sockets to the given interfaces, creates buffers, and starts three threads. One to receive frames, one to transform and filter, and one to send them. The threads communicate using shared memory and concepts of synchronization as mutexes and condition variables. One buffer is holding the incoming frames and one the outgoing. Both buffers work as queues using the *ConcurrentQueue* class. The ConcurrentQueue class is a wrapper of the queue class from the C++ standard library extending it with thread synchronization concepts.

As discussed in the evaluation chapter in the Memoria document, performance optimizations were required and therefore the ConcurentQueue class was replaced by the CircularBufferQueue class which internally uses the circular_buffer from the boost library. To allow the L2DirectCanEthernetGW and L2DirectEthernetCanGW classes to simply replace the ConcurentQueue buffers with a CircularBufferQueue buffer, a new interface *IQueue* was created with both queue classes inheriting from it. A UML class diagram showing the classes and their relations in a simplified way is shown in figure D.3

D.3.3 SDN Controller Setup

The setup depends on the chosen SDN controller which should provide documentation. For this work ONOS was used as SDN controller offering three different interfaces. A command-line interface, a graphical interface, and a REST API. It is also possible to develop and activate an SDN application to perform actions. Two examples to add flow-entries to a virtual switch are provided below. The first example shows how to upload JSON-formatted flows via the REST API from ONOS using curl. The second example does not use a real SDN controller and instead creates one flow-entry manually on the virtual switch.

Listing D.1: Example to upload json-formatted flows via the ONOS REST-API

```
sudo apt install curl
curl --user karaf:karaf -X POST --header 'Content-Type:\
application/json' --header 'Accept: application/json' -d \
@/home/<username>/<path-to-file>/<flowrule-file>.json \
'http://<ONOS-SDN-Controller-IP>:\
<ONOS-SDN-Controller-Port>/onos/v1/flows'
```

Listing D.2: Example to create a flow entry manually on a OVS virtual switch.

```
sudo ovs-ofctl add-flow <switchname> in_port=2, \
eth_type=0xeca1,actions=output:1,output:4
```

D.3.4 Gateway Setup

Before starting, terminal access to the gateway is required. Therefore a keyboard and a monitor can be connected to the gateway directly or, as done in this work, a SHH connection can be used. In the experimental setup used for this work the gateway has two network interfaces. The ethernet interface is used for the in-vehicle network and a WLAN interface is used for the SSH connection. To conduct certain experiments automatically, an SSH connection is necessary because commands need to be executed on different devices. In this work, the WLAN connection was also used to download the translator program, etc.

The first steps are related to the virtual switch software Open vSwitch which is very well documented. Open vSwitch needs to be started which happens automatically when having it installed as a service. A bridge, which is one switch instance, needs to be created within the virtual switch. The bridge needs to be configured with a network-wide unique datapath-id according to the network configuration. Also the switch needs to know to which SDN controller to connect and optionally which southbound APIs to support. Finally the switch needs to be enabled by the operating system. To show an exemplary way of doing this a code snippet of one of the used bash scripts is shown.

Listing D.3: Example configuration of a virtual switch

```
#e.g. $1=br1, $2=0000000000000002
ovs-vsctl add-br $1 \
 -- set bridge $1 other-config:datapath-id=$2 \
 -- set bridge $1 protocols=OpenFlow13 \
 -- set-controller $1 tcp:$controller
ip link set dev $1 up
```

To enable the CAN buses the used bitrate needs to be set. Different bitrates are supported as long as all connected devices are configured with the same value. To set up the CAN buses with a bitrate of 500 kbit/s, as applied in the in-vehicle network, the following command can be used.

Listing D.4: Example of configuring a CAN bus

```
#e.g. $1=can0
ip link add dev $1 type can bitrate 500000
ip link set dev $1 up
```

For each connected CAN bus it is necessary to create a pair of virtual ethernet interfaces. One interface needs to be connected to the switch and one to the translation instance. When connecting an interface to the switch it is important to assign an OpenFlow-port, since the host-given names for interfaces are ignored by OpenFlow, and being aware of the port identifiers is necessary for the network configuration. Again, a bash script snippet as an example is provided.

Listing D.5: Example adding virtual and non-virtual links to the virtual switch

```
#e.g. $1=br0; $2=veth0, $3=vETH0, $4=eth0
ip link add $2 type veth peer name $3
ip link set dev $2 up
ip link set dev $3 up
ovs-vsctl add-port $1 $2 -- set interface $2 \
        ofport_request=111 #veth -> goes to a CAN bus
ovs-vsctl add-port $1 $4 -- set interface $4 \
       ofport_request=222 #real ethernet
```

As the last step, the translating instances need to be started. For each CAN bus one instance needs to be started. When executing without or with invalid parameters a usage message is returned. The parameters which need to be set are -m –mapping, -c –caninterface, -e –ethinterface and -s –sourcemac. For the mapping *l2_direct* needs to be passed as an argument. For sourcemac, CAN and ethernet interfaces arguments according to the own network configuration needs to be passed, and depending on the permission configuration it is likely that superuser rights are required.

D.4 Program Compilation, Installation and Execution

The translator only needs to be compiled and started. To do so the project provides a CMake project file. So any CMake supporting IDE or any C++ compiler in combination with CMake as terminal program will be fine. The software was developed for Linux distributions and is not supported for

different OS. As mentioned in the CMake-file nlohmann_json and libboost are required libraries.

Listing D.6: Commands to compile the gateway project

```
sudo apt install nlohmann-json3-dev
sudo apt install libboost-all-dev
cmake .
cmake --build .
```

The virtual switch software OVS is necessary for this approach. To install the guide from https://docs.openvswitch.org/en/latest/intro/install/[1] can be followed. There are two options, installing it from binaries or building from source. In this work the software was built from source using v2.7.0. The following commands will install dependencies which are not installed by default on many Linux distributions and then download, compile and install Open vSwitch. The commands are tested using Ubuntu and could vary on the used package repositories.

Listing D.7: Commands to compile and install Open vSwitch, including dependecies

```
sudo apt install autoconf2.64
sudo apt install automake1.11
sudo apt install libtool-bin
sudo apt install gcc
# dependencies installed
git clone https://github.com/openvswitch/ovs.git
git checkout v2.7.0
cd ovs/
./boot.sh
./configure
make
sudo make install
# compiled and installed, not started yet
```

To be able to use OVS it is necessary to start the software. This can be done by configuring it as a system service or by starting it manually using the following commands.

Listing D.8: Commands to start Open vSwitch

```
sudo su
export PATH=$PATH:/usr/local/share/openvswitch/scripts
ovs-ctl start
```

```
exit
```

D.5 Tests

The testing process can be split into different phases. For functionality small pure virtual networks were used with network traffic generated using the command line. Those tests were used to confirm functionality in the developing process but the obtained results are just a self-validation and therefore not discussed in this document. This strategy was used to develop the translator program, to develop ONOS SDN applications, create flow-rule files, create scripts to upload those via the REST API and to configure the virtual switches.

With the functionality tested, the performance of the gateway was evaluated. The obtained results and the setup-details are documented in the Memoria document. Summarized, this test or experiment confirmed that the gateway approach could be used in real-time environments.

As the next step, less computational strong and therefore more realistic hardware was used to analyze timing behavior. To do so a Raspberry Pi as an embedded computer was used instead of a laptop. The Raspberry Pi was also connected to real CAN buses and real ethernet ports. One additional computer was used to generate traffic on the CAN bus and another computer to measure. The obtained results and the setup details are documented in detail in the Memoria document. To better analyze if the gateway could be permanently deployed in experimental setups, the measurements were done for many different configurations. To achieve this, a script executing commands on the sending computer, the measuring computer, and the gateway itself was used. Summarized, the gateway achieved to transmit traffic on maximal utilization for all possible CAN bandwidth configurations without loss and it was possible to determine time upper bounds.

Even though it was possible to determine time upper bounds, the previous experiment also showed that the forwarding process was slower than in the first experiment and grew when sending with higher bandwidth. This conclusion gave the motivation for speed optimization of the translator program and therefore a repetition of the second experiment.

The result of the improvements are shown in detail in in the Memoria document and can be seen as a success.

As a final test, the gateway was used to replace the existing gateway solution in the experimental setup used by the CoRE research group. First

one and then all of the used four gateways were replaced. Doing that, network configuration issues showed up and needed to be fixed first. Having one gateway successfully replaced, replacing the missing three worked without complications.

All the tests or experiments are documented in detail in the evaluation section in the Memoria document.

Appendix E

User Manual

E.1 Introduction

This work is about developing a gateway and not about user software. When an administrator sets the network and the gateway up correctly no further actions will be necessary.

A possible scenario having a user could be a mechanic with the need of reading all or certain CAN messages for control purposes. Therefore the administrator could preconfigure interfaces, which depend on the choice of which SDN controller is used. For example with ONOS as SDN controller, it is possible to create a graphical user interface which could be used to forward all the traffic to a certain device. For real deployment in a car, several security mechanisms would be needed to assure that a possible attacker would not be able to achieve the same. Having that in mind it is even more understandable that there are no "users" for the presented work.

E.2 Wiki Entry

This work serves a research group and therefore I was asked to create a wiki entry for other members of the group. Strictly speaking, these group members are not end users and the entry is mostly in German, but I think it might be interesting to see anyway.

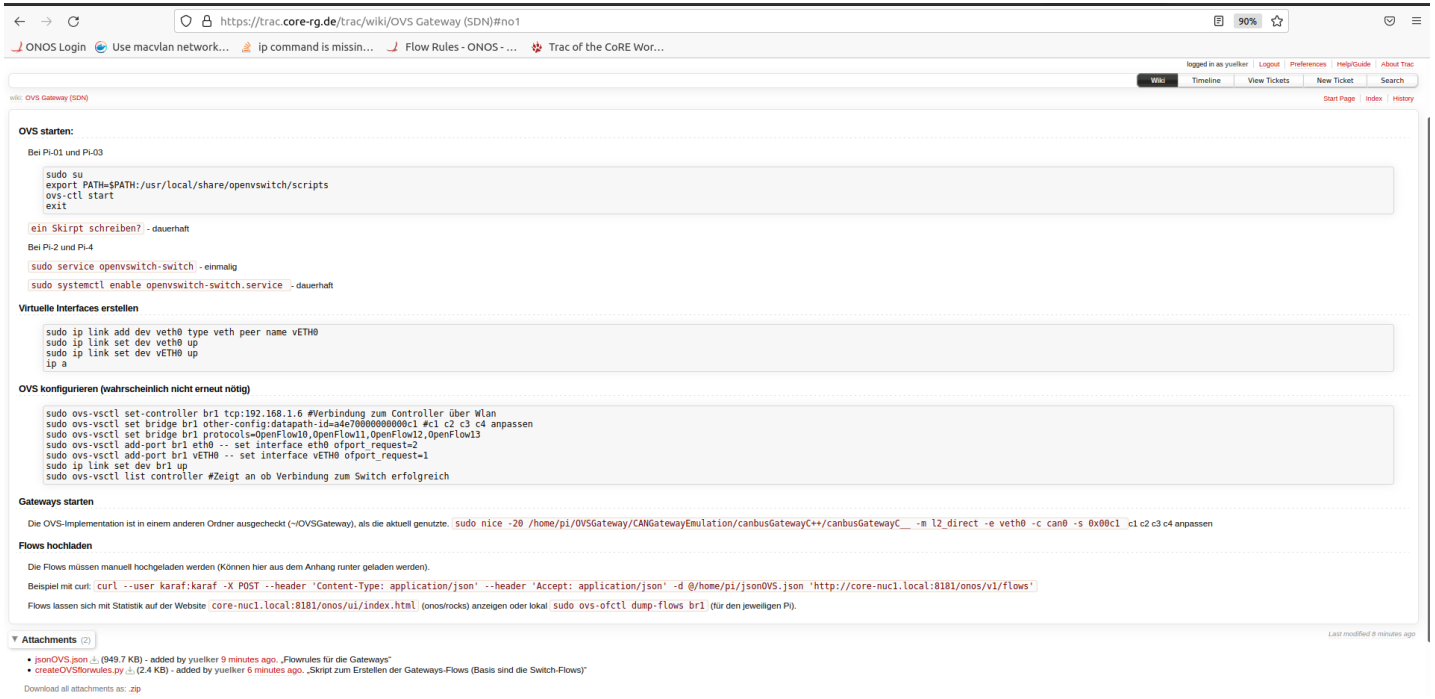


Figure E.1: Screenshot of the Wiki Entry

E.3 Hello World example

As explained above there are no real users, but a little starting help could be helpful to start the developed software and "play around a little bit". To do so with the lowest amount of effort this *helloworld.sh* can be used.

It installs and starts the needed software. It also creates a virtual switch, two virtual CAN interfaces and virtual ethernet links to connect them via the virtual switch. Then it configures the switch with a flow-rule and starts two translator instances. At the end of the script, software to monitor CAN traffic is started and some sample messages are sent. However, for better understanding, it is recommended to generate and monitor additional traffic after running the script.

```
sudo apt install autoconf2.64
sudo apt install automake1.11
sudo apt install libtool-bin
sudo apt install gcc
sudo apt install can-utils
sudo apt install gnome-terminal
sudo apt install nlohmann-json3-dev
sudo apt install libboost-all-dev
# dependencies installed

git clone https://github.com/openvswitch/ovs.git
git checkout v2.7.0
cd ./ovs/
./boot.sh
./configure
make
sudo make install -j4
sudo /sbin/modprobe openvswitch
 OVS compiled and installed

export PATH=$PATH:/usr/local/share/openvswitch/scripts
sudo env PATH=$PATH ovs-ctl start
#OVS started

sudo ip link add dev veth1 type veth peer name vETH1
sudo ip link set dev veth1 up
sudo ip link set dev vETH1 up
sudo ip link add dev veth2 type veth peer name vETH2
```

```
sudo ip link set dev veth2 up
sudo ip link set dev vETH2 up
#2x virtual eth created and enabled

sudo ip link add dev vcan1 type vcan
sudo ip link set dev vcan1 up
sudo ip link add dev vcan2 type vcan
sudo ip link set dev vcan2 up
#2x virtual can created and enabled

sudo ovs-vsctl add-br br1
sudo ovs-vsctl add-port br1 vETH1 -- set interface vETH1 \
        ofport_request=1
sudo ovs-vsctl add-port br1 vETH2 -- set interface vETH2 \
        ofport_request=2
sudo ip link set dev br1 up
#vETH1 and vETH2 connected to the enabled virtual switch
#(no SDN controller connected) - we create one flow manually

sudo ovs-ofctl del-flows br1 #delete old entries (if existing)
sudo ovs-ofctl add-flow br1 in_port=1,eth_type=0xeca1, \
        actions=output:2
#flow entry manually added
#(if ethertype=0xeca1 and IN_PORT=1 then forward to vETH2)

cd ../../canbusGatewayC++/ #<root folder of gateway software>
cmake .
cmake --build .
#gateway software compiled

sudo ./canbusGatewayC__ -m l2_direct -e \
        veth1 -c vcan1 -s 0x1234&
sudo ./canbusGatewayC__ -m l2_direct -e \
        veth2 -c vcan2 -s 0x2222&
#gateway software started (two terminals needed)

gnome-terminal -- candump vcan1 #own terminal
gnome-terminal -- candump vcan2 #own terminal
#use wireshark tcpdump or any other tool to monitor
#       the ethernet interfaces
#monitoring software started
```

```
cansend vcan1 123#C1C2AAAA
#both candump-terminals show C1C2AAAA

cansend vcan2 222#C2BBBB
#only candump2-terminal shows C2BBBB

sudo pkill canbusGatewayC_
#killing the gateway processes
cansend vcan1 123#C1CCCC
#only candump1-terminal show C1CCCC

#experiment, e.g. sudo ovs-ofctl dump-flows br1
```

Bibliography

[1] Linux Foundation. Installing open vswitch. https://docs.openvswitch.org/en/latest/intro/install/.