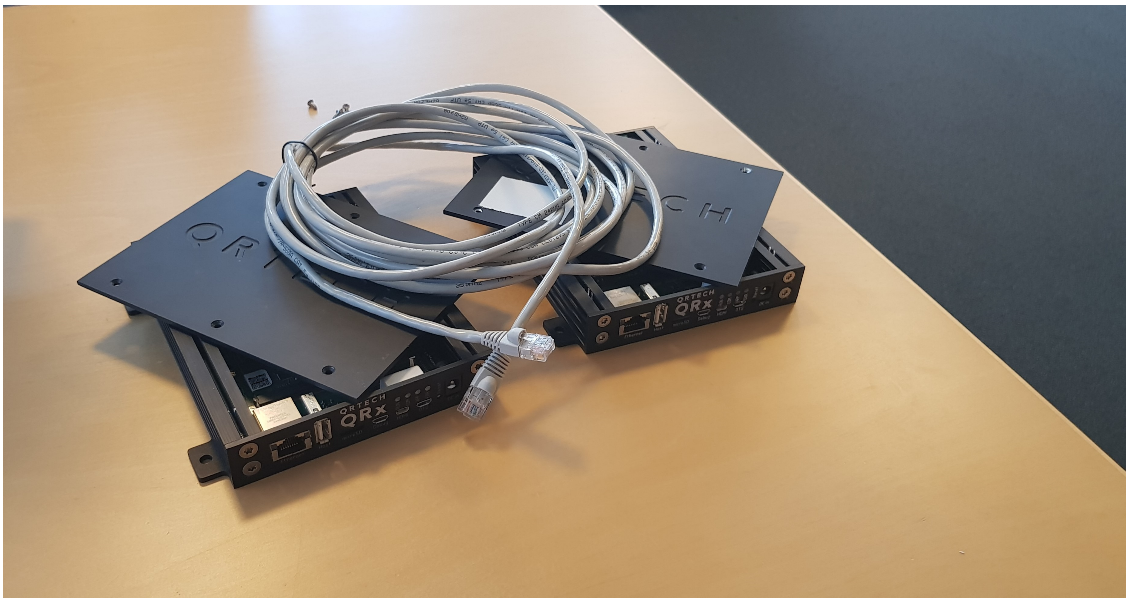
CHALMERS
UNIVERSITY OF TECHNOLOGY



The fast-lane development of Automotive Ethernet for Autonomous Drive

Master's thesis in Communication Engineering

Oscar Aspestrand
Viktor Claeson

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Master's thesis EX026/2018

The fast-lane development of Automotive Ethernet for Autonomous Drive

Oscar Aspestrand & Viktor Claeson

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2018

The fast-lane development of Automotive Ethernet for Autonomous Drive
Oscar Aspestrand
Viktor Claeson

Supervisors: Jonas Törnqvist, Qrtech
Alexander Polya, Qrtech
Chouaib Bencheikh Lehocine, Department of Signals and Systems
Examiner: Professor Alexandre Graell i Amat, Department of Electrical Engineering

Master's Thesis EX026/2018
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover:A photo of the AUTOSAR and Linux devices used in the communication link.

Typeset in LaTeX
Gothenburg, Sweden 2018

The fast-lane development of Automotive Ethernet for Autonomous Drive
Oscar Aspestrand & Viktor Claeson
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Autonomous drive has emerged as a new field of interest in recent years. The major car manufacturers are trying to implement the technologies needed for autonomous solutions, and the concept of Automotive Ethernet is being evaluated. It is of interest to investigate the current situation of the communication technologies present in vehicles today. How does the bus technologies cope with the increasing demands on bandwidth and how compatible are the different solutions. The purpose of this thesis is to answer these questions by discussing the relevant technologies in an Automotive Ethernet implementation. Focus lie on the construction of a small scale practical model aimed at mimicking a potential scenario where an Automotive Ethernet implementation is needed, in order to evaluate how feasible the current technologies are. The communication involves a node with the operating system AUTOSAR, common in vehicles today, and another node with Linux.

The results of the link evaluation were unexpectedly deficient and an investigation over the whole link was performed to localize the defective area. The implemented TCP/IP module in the AUTOSAR system displayed significant issues during large data transmissions. Small structural additions were implemented to the existing TCP/IP stack, which slightly improved the results, however the overall system remained significantly below capacity. In conclusion, the results indicate that there is still more work required for the relevant technologies before these implementations can be applied in practice. Further development suggestions are made, which imply that further research should be made on the upcoming Adaptive AUTOSAR platform as well as the Time-Sensitive Network protocol.

Keywords: Automotive Ethernet, AUTOSAR, CAN, FlexRay, LIN, MOST, SOME/IP, TSN.

Acknowledgements

We would like to start by thanking our examiner Prof. Alexandre Graell i Amat and supervisor Chouaib Bencheikh Lehocine at Chalmers for their support and feedback throughout the thesis.

Moreover, a great thank you to the people at QRTECH and especially our supervisors, Jonas Törnqvist and Alexander Polya, as well as Joakim Plate and Joakim Hesselgren for support and guidance. We have learned a lot from you during this project and all your help has been very appreciated. In addition, we would like to thank QRTECH for the opportunity to carry out our thesis and for supplying us with the equipment, work space and breakfast to perform this project.

Oscar Aspestrand & Viktor Claeson, Gothenburg, June 2018

Contents

List of Figures

List of Tables

List of Acronyms

AUTOSAR	AUTomotive Open System ARchitecture
AVB	Audio Video Bridging
BE	Best Effort
BMCA	Best Master Clock Algorithm
BSW	Basic Software
CAN	Controller Area Network
CBS	Credit Based Shaper
CNC	Centralized Network Controller
CRC	Cyclic Redundancy Check
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
CUC	Centralized User Configuration
E/E	Electrics and Electronics
ECU	Electronic Control Units
FCS	Frame Check Sequence
IFG	Inter Frame Gap
LDF	LIN Description File
LDPP	Link Layer Discovery Protocol
LIN	Local Interconnect Network
lwIP	lightweight IP
MAC	Medium Access Control
MOST	Media Oriented Systems Transport
NIC	Network Interface Controller
OEM	Original Equipment Manager
OSI	Open System Interconnection
P2P	Point-to-Point
PDU	Protocol Data Unit
PTP	Precision Time Protocol
Pub/Sub	Publish/Subscribe
QoS	Quality of Service
RPC	Remote Procedure Call
RTE	RunTime Environment
RTT	Round Trip Time
SD	Service Discovery
SOME/IP	Scalable service-Oriented MiddlewarE over IP
SRP	Stream Reserve Protocol
SWC	SoftWare Component
TAS	Time Aware Shaper

TDMA	Time Division Multiple Access
TSA	Transmission Selection Algorithm
TSN	Time-Sensitive Network
TT	Time-Triggered
VFB	Virtual Function Bus
VID	VLAN Identifier
VLAN	Virtual Local Area Network

1
Introduction

In the modern day automotive industry, the increasing implementations of new applications and infotainment systems implies that a large amount of Electronic Control Units (ECUs) are required within a vehicle. Comparing contemporary automobiles in the industry to former versions from 30-40 years ago, the increase in electronic equipment is immense. Vehicles today include ECUs for a variety of features, including aspects such as safety as well as comfort. With some of the ECUs connected to crucial systems, the communication system in a vehicle must not falter. Thus, it must be a high-speed-high-reliability system. As there exist many well-developed communication protocols for other fields of expertise, an approach has been to adapt these to fit in the automotive industry in combination with specifically constructing some protocols for the automotive industry. With the expanding system complexity, the need for a standardized architecture when implementing communication systems in vehicles is vital. The AUTomotive Open System ARchitecture (AUTOSAR) Development Partnership was launched in 2003 by leading automotive manufacturers and suppliers and defines a standardized architecture for communication systems in vehicles, with the aim of keeping the application software independent of the hardware [1], [2].

Furthermore, ECUs running on different operating systems may lack the possibility to communicate without a compatibility extension. Scalable service-Oriented MiddlewarE over IP (SOME/IP) is a protocol developed to function as a middleware solution for the automotive industry and allows ECUs with different operating systems to communicate. The protocol includes serialization, which transforms application data into a stream of bytes instead of system specific structured data [3]. In addition, SOME/IP middleware shifts the communication concept from a signal-oriented to a service-oriented. This means that information is sent when the receiver finds a need for specific data, in contrast to signal-oriented transmissions where information is sent when the sender finds a need e.g. when a value is changed or updated [4].

The AUTOSAR systems in the industry today mostly rely on bus systems like CAN, LIN or FlexRay to transport data between ECUs in the vehicle. These bus type systems rely on a broadcast data concept. This means that when one ECU has data to transmit, the content is uploaded to the bus, where any receiving ECU can pick it up. However, the data occupy the bus for all other ECUs even though they do not need that data. This can cause overload issues, especially since the bus systems were originally designed to transmit small data packets. The sensors and infotainment implemented in vehicles today includes communication of a larger amount of data, which was not a concept in the initial bus link idea. An example of this could be how a constant flow of LIDAR data should be sent to a neural network which then can establish objects in front of a vehicle [5]. The data constraints has led to an initiative of implementing Ethernet to automotives, which is a well functioning protocol in other areas and it can handle large amounts of data. In vehicles however, the requirements on speed and reliability are more crucial as communication failures could lead to severe consequences [6], [7].

The implementation of Ethernet usually also implies that TCP/IP, in accordance with the OSI-model, is used. This is one of the issues that the implementation of Automotive Ethernet faces, since TCP/IP does not generally provide high-speed-high-reliability communication. An example is that TCP/IP employs concept such as retransmission if a packet was not received correctly. That scenario in a vehicle could lead to fatal consequences if ECUs in charge of safety features do not transmit/receive the packet immediately or if it is corrupted. An alternative to TCP/IP and the OSI stack is to employ Ethernet extensions such as Audio Video Bridging / Time-Sensitive Network (AVB/TSN). As the data is not supposed to be transported across network boundaries in automotives, AVB/TSN protocols can provide sufficient communication using only Ethernet frames. The frames utilize the VLAN tag to prioritize the TSN frames and thereby AVB/TSN is able to provide low-latency and high-quality transmission of streaming data [4].

1.1 Purpose

The core purpose of this thesis is an in-depth investigation of how far the automotive industry has come in terms of high-speed-high-reliability communication technologies. Specifically, investigate Automotive Ethernet and evaluate whether it can be a solution for autonomous drive applications. The focus lie on the construction of a small scale test system, which is developed with the purpose of providing measurement results and context to the theoretical background.

1.2 Scope

As AUTOSAR is continuously in development, improvements could have been made with every new version and therefore the results may be version related. The version used during the project was the AUTOSAR 4.2.2 release. In this version SOME/IP is not implemented as a module yet, thus only a literature and feasibility study will be performed. To implement SOME/IP as a complex driver was under consideration but was deemed too time demanding and complicated for the scope of this thesis. Likewise, to update the existing AUTOSAR available at the company was considered and rejected due to time constraints.

Furthermore, in the designed practical model, the Ethernet cables and version used was the regular 100Base-Tx instead of the BroadR-Reach thought-out for Automotive Ethernet. This is due to equipment and cost restrictions at the company. Similarly, for the comparison of different communication technologies, only Ethernet will be implemented in practice. The discussion and comparison to the other technologies will be theoretical and utilizing previous work within the field.

1.3 Related Work

In [8], the authors successfully implemented a running TCP/IP Ethernet connection on an AUTOSAR platform. The microcontroller used (TMS570LS3137) was less powerful than the one used in this thesis (MPC5744P). However, their goal was to create a running web application which does not require quantities of either memory or execution speed. In addition, different development tools and protocol versions where used compared to this project, such as the TCP/IP stack and AUTOSAR configuration.

Degermark et al. [9] used a single-core 32-bit processor, Pentium Pro, with the same clock speed of 200MHz as the processor used in this project. They managed to perform quick IP routing lookups of forwarding tables at gigabit speeds, hence our processor should be capable of a desirable bit rate hardware-wise.

1.4 Thesis Outline

The structure for the remainder of this thesis is organized as follows. In Chapter 2, some background theory is presented regarding important concepts of Automotive Ethernet. Chapter 3 covers the implementation choices and methods utilized when designing the communication link. The result of the practical model when transmitting images over the link is presented in Chapter 4. Chapter 5 includes a discussion of the topics included in this thesis as well as further analysis of the results of the communication link. Finally, Chapter 6 summarizes the conclusions and suggestions for future improvements.

2
Background Theory for Communication Technologies in Automotives

This chapter aims at introducing the reader to information about the communication technologies and concepts involved in the Automotive Ethernet development. Firstly, some preliminaries are briefly explained.

2.1 Preliminaries

The following concepts were deemed important to be familiar with, as they are referenced in the thesis and therefore a brief description was included.

2.1.1 OSI-model

The Open System Interconnection (OSI) model is a conceptual framework that alleviates terminology and complex interactions within computer networks. It consists of 7 layers ranging from an application software down to the physical interface on the hardware, Table 2.1 display the different layers along with a short description. The communication flow goes from the higher levels to the lower ones as data is passed on through the network. The OSI-model is often used as a reference for protocols to determine at what abstraction level they operate e.g. TCP is a layer 4 protocol and Ethernet a layer 1 protocol.

Table 2.1: A representation of the OSI-model, including a short description for each of the 7 layers.

7	Application	Network connection to the application, providing services.
6	Presentation	Data conversion to different representations and encryption.
5	Session	Managing connection sessions, e.g. open, close and recover.
4	Transport	End-to-end transmission of data.
3	Network	Packet forwarding and routing scheme.
2	Data Link	Transfer data between nodes within one network.
1	Physical	Connection to the physical transmission medium.

2.1.2 Cyclic Redundancy Check

Cyclic Redundancy Check (CRC) is an error-detecting method, which uses cyclic codes to control if raw data has been corrupted. It includes a data verification to the original raw bytes, on which the receiving side evaluates if the data is correct or not.

The CRC operation can mathematically be explained as a binary data word being treated as a polynomial, with each polynomial coefficient being zero or one, and divided by a generator polynomial often referred to as a CRC polynomial. The remainder of the division is the data verification check of the original raw bytes. It is sent as a Frame Check Sequence (FCS) together with the raw bytes (as a trailer). On the receiving side, the CRC is repeated and the remainder is compared to the sent FCS. If they match, the result is most likely correct. There is a chance that there was just the right amount of bit errors to go undetected by the CRC. This is due to the CRC polynomial used, as each only have a limited amount of flipped bits it can provide protection against.

An example of how a CRC is calculated can be seen in Figure 2.1. The divisor is a polynomial which is predefined in each device and is one bit larger than the remainder. The divisor performs an XOR operation in a sliding window-manner on the whole payload step-wise. To start with, on the transmission side, the payload is right-padded with zeros equal to the amount of bits of the desired remainder, and then the bitwise XOR operation with the divisor starts. The division process continues until the payload, without the padding, is a zero-vector, leaving only the padding bits as the remainder. Upon arrival at the receiver, the payload and CRC trailer is extracted and the same operation is repeated, with the same divisor as the transmitter used. However, in this case, the payload is right-padded with the remainder instead of zeros. If the final result is zero, the message is most likely correct.

There are several methods of constructing a polynomial, which is dependent on the application. Some provide good protection for long data words and some the opposite. One of the most commonly used is the IEEE CRC-32 polynomial, where the polynomial consist of 33 bits and the remainder is 32 bits [10].

```
Transmitter:
Payload: 100100100011110 , Divisor: 1011 (CRC-3-GSM, used in mobile
    networks)

100100100011110 000   <- zero pad, will become remainder.
1011                  <- Divisor.
001000100011110 000
  1011        <-2nd bit = 0, therefore moving on to next 1 in payload
000011100011110 000         ... in order to align the divisor.
    1011
000001010011110 000
     1011
000000001011110 000
        1011
000000000000110 000
        101 1
000000000000011 100
       10 11
000000000000001 010
              1 011
000000000000000 001 <- 001 will be the remainder, hence the CRC check.

Receiver:
Payload: 100100100011110 , Divisor: 1011 (CRC-3-GSM, used in mobile
    networks)
Received CRC check: 001

100100100011110 001   <- Right pad with the CRC check.
1011                  <- Divisor known in receiver.
001000100011110 001
  1011
000011100011110 001
    1011
000001010011110 001
     1011
000000001011110 001
        1011
000000000000110 001
            101 1
000000000000011 101
             10 11
000000000000001 011
              1 011
000000000000000 000   <- A zero vector shows correct message.
```

Figure 2.1: Transmitter and receiver CRC algorithm

2.1.3 TCP Socket Communication

TCP/IP communication is built on the foundation of so-called socket communication. A TCP socket is an internal representation of an endpoint connected with the local IP address and a port number. A socket can be defined as one out of two roles, either as a server socket or as a client socket. The former binds a socket descriptor to a specific port number and defines for what type of connections it should handle e.g. TCP or UDP. The server is then set to a listening mode where it awaits connections from client sockets, whom it will accept or reject depending on if they fulfill the correct specifications. If a connection is accepted, the socket pair can now begin to send and receive data.

The client socket does not require a binding to specific port number, it will receive the first available one. The client socket must attempt a connect call towards an ip-endpoint (server socket), specifying what type of connection that should be established and await an accept. This process can be seen in Figure 2.2. The types of socket are called: Datagram sockets, Stream sockets and Raw sockets. Datagram sockets are also-called connectionless sockets and are used by UDP, in contrast to Stream sockets which are called connection-oriented sockets and are used by TCP. A raw socket simply encapsulates data without any formatting from the transport layer.

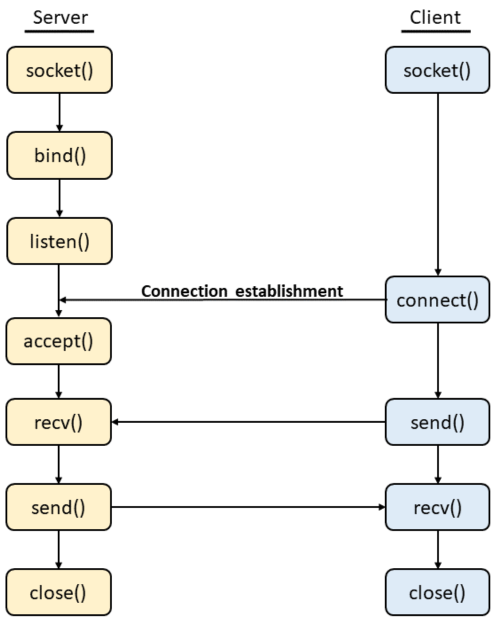


Figure 2.2: Socket communication procedure between a server and a client socket.

2.1.4 TCP Delay Efficiency

Most TCP stacks employ the so-called Nagle algorithm, which enforces the send command to wait for a maximum segment sized packet before transmission. This behaviour is only interrupted when an acknowledgement (ACK) is received on the last remaining bytes in flight, i.e. so far un-acked bytes. The send buffer is then transmitted regardless of the filled segment size. The Nagle algorithm is in most cases an effective way to utilize the bandwidth and to avoid congestion problems, by not transmitting packets with a small payload and large overhead. However, it might not be suitable for all types of systems, one example being time-critical real-time services. In these systems, latency is of highest importance and the large overhead cost might be insignificant compared to the transmission delay when waiting for maximum segment sized packets.

An issue that can occur with Nagle's algorithm is when it is paired with the so-called TCP delayed acknowledgements. This is also a delay efficiency method that most TCP stacks utilize, which allows the receiving application to respond with less than one ACK per segment. Furthermore, it can respond with a window update and an eventual immediate response alongside the ACK. However, when paired with the Nagle algorithm, there are scenarios where they inhibit each other. An example of this is if the receiving end does not respond with an ACK due to the whole package not being received yet. At the same time, the Nagle algorithm is hindering the transmission of the last remaining bytes of the package because it does not fill the maximum segment size. This stalls the communication for the duration of the timeout on the ACK [11].

2.2 Bus technologies in automotives

Bus-communication is the standard for data exchange in real-time between ECUs in automotives. For serial data exchange between ECUs in automotive applications only the lower layers of the OSI model are necessary, i.e. the Data Link Layer and Physical Layer, in combination with the Application Layer. The reduction of the original seven-layer OSI stack simplifies and improves the speed of the communication. Bus type communication relies on a broadcast type concept, where if a node wants to communicate it transmits data to the bus and any other node can receive the message. Even if there is a specific node that requires the data, the message is still allocated on the bus for all other nodes as well. This can be a source of congestion problems at high data loads [12]. There exists a few different bus type protocols used for different applications, the most common in vehicles are CAN, FlexRay, LIN and MOST (see Sections 2.2.1-2.2.4). The different network technologies must all provide solutions to the fundamentals of sharing a serial interface with several users, organizing the access to the medium, provide a certain data rate and a robust transmission.

2.2.1 CAN

The Controller Area Network (CAN) protocol, common in vehicles today, was developed to allow microcontrollers to communicate in automotive applications. It was developed by Robert Bosch in 1983 and it later became a standard for serial data exchange in real-time between ECUs in automotives. CAN is a distributed communication system organized as a hierarchy, which operates using a simplified OSI stack mainly relying on the physical and data link layer. Instead of including source or destination addresses in the messages transmitted like e.g. in Ethernet, each message contains an identifier which specifies the message priority. As a transmitted message is available for any node on the bus, each node performs an individual test to determine if the message is of importance and should be accepted or not. Every node on the bus responds with an acknowledgement on an error-free CAN frame by setting a dominant bit in the acknowledgement space (see Table 2.2) of the same packet. The receiving nodes do this regardless to if they actually use the data or not. The transmitting node recognizes an acknowledgement without knowing from which receiving node it originated from. Adding that with the fact that one acknowledgement is sufficient for the transmitter to perceive it as a successful transmission, it causes a source of uncertainty as there is no proof that the intended receiver node has seen the packet.

The hierarchy of the CAN network relies on the message priorities determined by the identifiers. The method to determine which identifier is more important than the other is called arbitration and it distinguishes between dominant (0) and recessive (1) bits in the Message-IDentifiers. In the case of two ECUs starting to transmit at the same time, the ECU whose message starts with the largest amount of dominant bits is determined as the higher priority. In other words, a lower Message-ID implies a higher priority. The node with a higher Message-ID, and thereby a lower message priority, must wait a given fallback time before it can try to access the bus again. However, if the CAN-bus is idle then any node can start transmitting .

The control over the transmission media is a combination of CSMA/CD (Carrier Sense Multiple Access with Collision Detection) and NBA (Non-destructive Bitwise Arbitration). This combination enables the maximum use of a CAN bus data transfer capability. The CAN system provides high reliability, flexibility and robustness in harsh environments such as within a vehicle. However, it was constructed to handle small data loads and it can only achieve data rates of 1 Mbps (High-Speed CAN). This means that it cannot handle the high data loads that some of the new applications require. There can also be issues regarding congestion control, as higher priority messages can block the bus for long periods of time if the data load is large [7], [13], [14], [15].

Table 2.2: Structure of a CAN frame.

Field name	**Length [bits]**	**Details**
Start of frame (SOF)	1	Indicates the start of a frame transmission and is always a 0.
Identifier (A)	11	First part of the unique identifier, which also indicates the message priority.
Remote Transmission Request (RTR) or Substitute Remote Request (SRR)	1	RTR must be dominant (0) for data frames and recessive (1) for so-called Remote Frames. SRR must be recessive (1).
Identifier extension bit (IDE)	1	With 11-bit identifiers it must be dominant (0) and for 29-bit identifiers it must be recessive (1) and then includes the two following fields.
Identifier (B)	18	Second part of the unique identifier, which also indicates the message priority.
Remote Transmission Request (RTR)	1	RTR must be dominant (0) for data frames and recessive (1) for so-called 'Remote Frames'.
Reserved bits	1 or 2	One or two reserved bits depending on 11-bit or 29-bit identifier case.
Data length code (DLC)	4	Specifies the length of data field (0-8 bytes).
Data field	0-64	The payload of the frame (length according to the DLC field).
CRC	15	The cyclic redundancy check.
CRC delimiter	1	Must be recessive (1).
ACK slot	1	Is set to recessive (1) by the transmitter and all receiving nodes set this to dominant (0) if they have received the frame error-free.
ACK delimiter	1	Must be recessive (1).
End of frame (EOF)	7	Must be recessive (1).

2.2.2 FlexRay

As the demand for communication required larger amounts of data being transmitted from an increasing amount of ECUs, the CAN standard did not meet the requirements. The FlexRay serial communication protocol was developed for data exchange in more safety-critical automotive applications, with higher demands on reliability and safety of data as well as delivering 10 Mbps compared to CAN with 1 Mbps. One of the most significant differences, compared to CAN, is that FlexRay uses a Time Division Multiple Access (TDMA) scheme for the Medium Access Control (MAC). Instead of sensing the medium and terminate transmission on noticing a busy bus, TDMA provides each ECU in the closed network with time slots which correlates to the time-triggered communication architecture of FlexRay. The architecture's core property of being time-triggered provides a static activation of actions on each cycle, which typically is between one to five milliseconds. FlexRay was supposed to be a flexible network capable of asynchronous operations, which a closed TDMA network does not supply. Hence, FlexRay networks provide a dynamic segment in combination with the static segment. A FlexRay cycle containing these segments is portrayed in Figure 2.3. Besides the static and dynamic segments, there is also a segment called Symbol Window. It is primarily used for maintenance and to identify special cycles (such as a *cold-start*). Furthermore, between two cycles is a Network Idle Time segment which is a predefined interruption utilized to adjust for potential drifts in the system [13], [16].

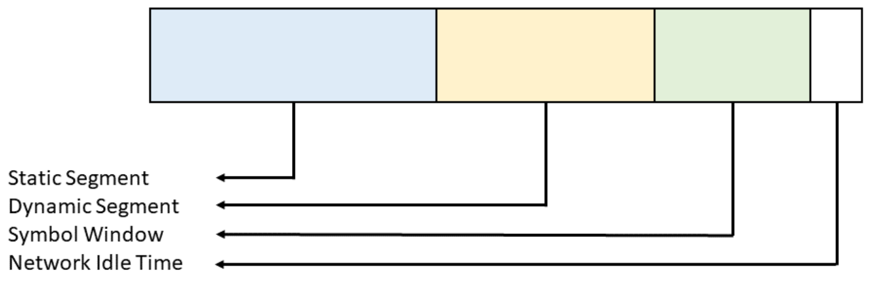


Figure 2.3: Visual representation of one FlexRay duty cycle, where the cycle is typically 1-5 ms. It consists of four segments, where the Network Idle Time is a predefined interruption between cycles to adjust for any potential drifts.

The static segment is divided into several slots, correlating to an ECU in the network. To avoid slowing down the FlexRay cycle by adding more static slots, the dynamic segment can allow transmission of the less critical data. The dynamic segment is also divided into slots, so-called minislots, which are prioritized according to Frame IDs. The segment is of a fixed length, limiting the data that can be transmitted per cycle. Furthermore, each minislot is of a configurable duration that is typically a so-called macrotick (microsecond) long. A macrotick is the smallest unit of time within a FlexRay network and is synchronized on every node. If the ECU corresponding to a minislot is not ready for transmission, the time slot in the dynamic segment is lost and the next minislot can take its turn. When an ECU decides to broadcast, all future minislots are set on hold until the broadcast is completed. If the dynamic segment window is filled before all ECUs minislots have been handled, these have to wait until the next cycle before they have an opportunity to broadcast. This behaviour of the dynamic segment is similar to the event-triggered behaviour in a CAN bus, by allowing nodes access to the bus in a prioritized fashion if it is available [16].

In order to maintain the schedule, each ECUs clock must be synchronized. This is done by the static segment of FlexRay and at least two nodes are required. In order to initialize a FlexRay clock synchronization at startup, certain startup frames are transmitted from specific nodes. This process is called a *cold-start* and the nodes responsible for sending the startup frames are called cold-start nodes. Upon completion of the cold-start, two other nodes are preconfigured to broadcast special sync frames while the other nodes are idle. This process will synchronize each node's internal oscillator to the network's tick. A clock must not exceed an offset of 0.15 % of the reference clock. Due to e.g. frequency differences between ECUs, two nodes' clocks may initially be the same but drift apart over time and thereby resulting in a maximally allowed drift of 0.3 %. To avoid such an event, the Network Idle Time slot adjusts the clock accordingly to the drift of the previous cycle.

A FlexRay frame consists of a 5-byte header, between 0-254 bytes of payload and a 3-byte trailer. The content of the frame structure can be seen in Table 2.3. The payload length is at maximum 254 bytes, which is about 30 times the size of a CAN payload length [12], [13].

Table 2.3: Structure of a FlexRay frame.

Field name	**Length [bits]**	**Details**
Reserved bit	1	Reserved bit
Payload preamble indicator	1	Indicates whether the data packet contains a payload.
Null frame indicator	1	Indicates whether the data packet is a Null frame.
Sync frame indicator	1	Indicates whether the data packet is a Sync frame.
Startup frame indicator	1	Indicates whether the data packet is a Startup frame.
Frame ID	11	Packet identifier.
Payload length	7	Length indicator for the amount of bytes in the payload.
Header CRC	11	CRC for header. Covers the Null frame indicator to Payload length.
Cycle count	6	Indicator for current cycle.
Data field (payload)	0-254 bytes	The actual payload of the frame. Transmitter data is signaled by a "1" and a receiver NACK by a "0".
Payload CRC	24	CRC of the payload.

2.2.3 LIN

The Local Interconnect Network (LIN) protocol is used for automotive applications such as power windows, central locks etc. (where CAN would overperform). These are applications that do not have such high demands on being time or safety critical. It was a commonly deployed standard accepted in 2002, created by the LIN consortium which was initiated in 1998. It is a single-ended system that is designed so that 16 ECUs can share the media that the bus provides, reaching speeds of up to 20 kbps. However, the most important feature of LIN is that it should be cost efficient, high data rates were never the key requirement.

The multi-user access of the LIN bus is built on a Master/Slave configuration, with one Master and up to 15 Slaves. The slaves are only allowed to transmit after they have been polled with a header by the master. However, as it is a bus system, two slaves can also engage in a master initiated communication and potentially all information on the bus can be read by any unit attached to it [13].

A LIN transmission occurs during a LIN frame slot, which consists of a Message Header, Message Response and a response space. The header is transmitted to the bus by the master node and the response by the slave after a short processing time called the response space, as can be seen in Figure 2.4. The master node can itself also act as a slave in the exchange of data, as it contains a slave task in parallel with the master task.

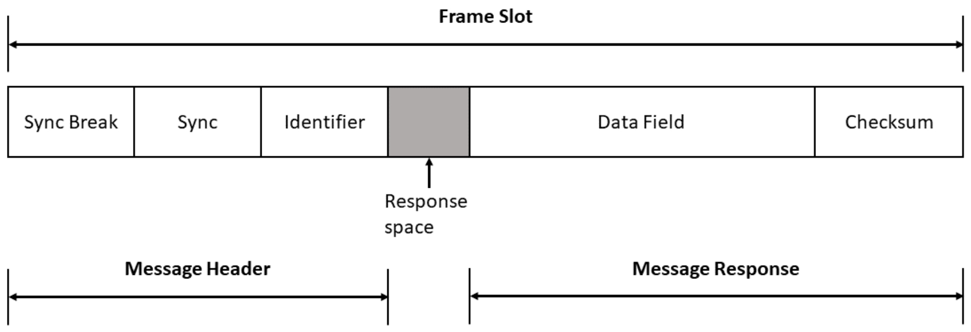


Figure 2.4: Structure of a LIN frame slot, with both the Master node's Message Header and the Slave node's Message Response.

The Message Header consists of three fields: A Sync Break field, Sync field and an Identifier field. The break field consists of 13 dominant bits (0) and a recessive bit delimiter, with the purpose of announcing to all nodes that a message is incoming. Following is the sync field which allows the slave to determine the transmission rate that the master uses by calculating the time between two falling edges in the 8-bit pattern of "0x55" (01010101). The slave can then synchronize its internal baud rate to match the bus. The identifier field consists of 6 ID bits and 2 parity bits. It is using the identifier field that each node can determine if they are a publisher or subscriber to the specific identifier. Each LIN bus has 64 IDs, of which 60 are used for carrying data and the remaining are for diagnostics, extensions and protocol enhancements.

The Message Response is divided into two fields, namely a Data field and a Checksum field. Based on the identifier field of the header, a slave will recognize that it has been addressed and put its response in the data field. The data field contains 1-8 bytes and the following checksum is of 8 bits. The checksum algorithms can vary between different versions of LIN, but the classic is performed by simply summing the data bytes and the ID.

To configure the system, a so-called LIN description file (LDF) is used. The LDF defines on which ID each node will act, what actions should be executed, the baud rate and delays. After the LDF has been processed by a system generator, the master node immediately begins its procedure of sending headers. An additional mechanism of putting nodes to sleep in order to save power may also be implemented. However, this action is mainly active during diagnostics of the network or if the whole LIN bus has been inactive for more than four seconds [17], [18].

2.2.4 MOST

For automotive applications requiring larger bandwidth, which is mostly infotainment applications, the bus most commonly used is the Media Oriented Systems Transport (MOST). The MOST corporation was founded in 1998 and MOST is an optical data bus technology that can achieve data rates up to 150 Mbps. The MOST technology provides a synchronous transmission of audio and video data as well as specifying interfaces and functions for infotainment applications at a high abstraction level and is optimized for streaming data. However, unlike previously mentioned technologies, MOST is not only involving the Physical and Data Link layer but the whole 7-layer OSI reference model. MOST may be used for a variety of bit rates, depending on the application, and are available in three different generations (MOST25, MOST50, MOST150) that ranges from 25 to 150 Mbps. Higher bit rates may be available, but are not of official release [19], [20], [21].

2.2.4.1 System Description

A MOST network topology is based on a ring, either an actual or a virtual, which can consist of up to 64 MOST devices. It is a synchronous network, with one MOST device acting as a Timing Master that continuously sends the preamble that every frame begins with. The other nodes in the network can then use this to synchronize their clocks. The network itself most commonly follows a one-directed ring topology, but may also be of star topology for specific implementations. Due to the 7-layered OSI implementation, a MOST system can simply add and remove nodes on demand. In order to configure each ECU within the network, MOST offers a channel dedicated to control messages. The other two channels available to a MOST application are a synchronous and an asynchronous channel for data transmission [13], [22].

The architecture of a MOST device consists of three main parts, namely a Physical Interface, Network Services and so-called Function Blocks (FBlocks). The architecture in contrast to the OSI-model is visualized in Figure 2.5. The physical interface that interacts with the hardware can be either optical or electrical, depending on the MOST generation (where MOST25 is only optical). The MOST Network Interface Controller (NIC) manages the different network services in the MOST architecture and controls the access to the three different channels. The FBlocks are the interfaces to the available function and services of a device. Devices may offer these services towards an application or to the MOST network. A FBlock can be defined as different types, namely as a Controller, Slave or Human Machine Interface (HMI). Controllers control one or more FBlocks that are of the type Slave, which has no information about the network. An example of a controller would be the *Netblock*, which is responsible for the administration of a device (e.g. has a list of all the functions and manages all the addresses of the device). HMIs can be compared to a User Interface (UI), as they are used for interaction between the user and the devices [23].

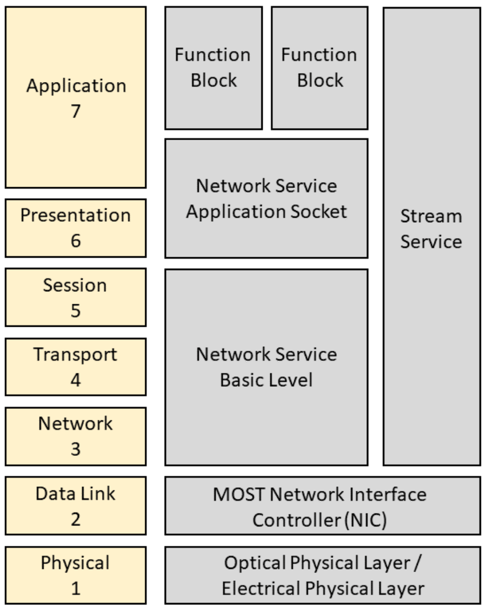

Application 7
Presentation 6
Session 5
Transport 4
Network 3
Data Link 2
Physical 1
Function Block
Function Block
Network Service Application Socket
Network Service Basic Level
Stream Service
MOST Network Interface Controller (NIC)
Optical Physical Layer / Electrical Physical Layer

Figure 2.5: Visual representation of the MOST architecture (in gray) in contrast to the OSI-model (in yellow).

2.2.4.2 Frame Format

A MOST message could be distributed over several MOST frames, which represent the constantly repeated structure in which the MOST traffic is organized. The three channels are represented in the frame structure, as one frame contains the respective channel for a synchronous transmission of streaming data, an asynchronous transmission of packet data and the transmission of control data. A visual representation of this can be seen in Figure 2.6. The frame format also depends on what MOST generation that is used, as they significantly differ in available sizes. A MOST25 frame consists of 64 bytes, a MOST50 frame the double i.e. 128 bytes due to the doubled bandwidth and likewise a MOST150 frame consists of 384 bytes. Out of these frame sizes, the channel dedicated for control messages take up 2 bytes for MOST25 and 4 bytes for MOST50 and MOST150 out of each frame [23].

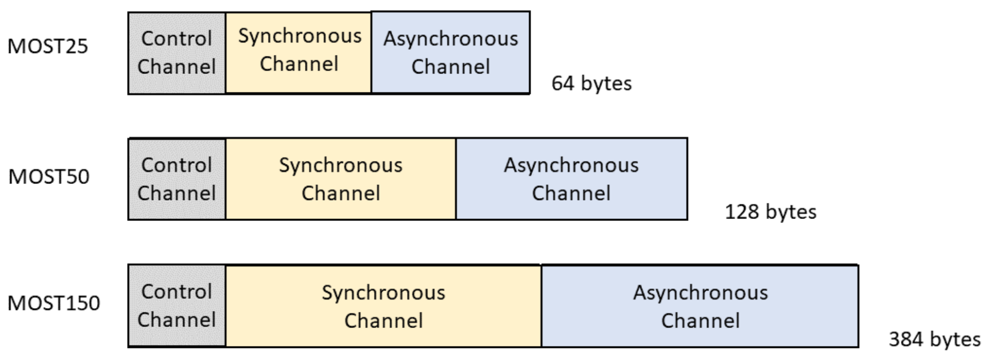


Figure 2.6: Visual representation of a MOST frame for the different generations. Each frame has dedicated bytes for control messages (2 bytes for MOST25 and 4 bytes for MOST50 & MOST150) and the remaining bytes are split between the asynchronous and synchronous channels.

2.3 Background of Automotive Ethernet

The incentive to adopt Ethernet in the automotive industry is to prepare for the increasing bandwidth demands of the future. More complex systems will complement the many ECUs already present in vehicles today, and with them comes even further requirements. The high performing technologies implemented in vehicles today, such as FlexRay and MOST, are expensive and complex as well as unable to reach the exponentially increasing bandwidth demands. However, Ethernet networks have proven to be efficient in other areas and could be the solution to the bandwidth demands now required in the automotive industry [24].

2.3.1 Switched network background

The majority of Ethernet networks installed today does not rely on the CSMA/CD mechanism of the original IEEE Ethernet. CSMA/CD was based on the ALOHA method for multi-user access, which simply involves retransmissions in case of collisions, with the addition of establishing when the channel is occupied as well as an exponential random back-off period in case of collisions. Switched networks with Point-to-Point (P2P) links was the next step for Ethernet networks, making CSMA/CD somewhat obsolete. Two units PHYs will be directly connected and packets will traverse according to the addresses established between the PHYs. The communication in switched networks is called full duplex and it is controlled by a MAC which provides a mechanism to decide when packets should be sent. This, in turn, enables flow control and allows for limited resource consumption in terms of buffering and switching bandwidth. Each Ethernet interface has a unique serial number assigned to it, known as the MAC address, which consists of 48 bits. The MAC address is used in the switched network to determine if a node should read the packets full content or simply forward it, i.e. if the destination address of the Ethernet frame matches the node's MAC address [13].

Ethernet has been adapted to many different areas and industries, such as aviation and telecommunications, all of which has some adaptations to the original IEEE Ethernet in order to match their respective restrictions. The automotive industry is no different, it would like to reuse as much of existing technologies as possible.

2.3.2 BroadR-Reach

BroadR-Reach is an Ethernet standard on the physical layer, which reduces connectivity loss and cable weight. It utilizes an unshielded single twisted pair cable that together with the IEEE 802.3 standard reliably deliver up to 100 Mbps. The cable should not exceed over 15 meters due to the high possibility of electromagnetic distortion in vehicles. Compared to IEEE 1000Base-T and IEEE 100TX, which uses 65-80 MHz bandwidth, Broad-R Reach utilizes only 33.3 MHz but can deliver up to 100 Mbps as a result of the high spectral efficiency gained from encoding techniques in the Physical Layer [25]. The standard can incorporate multiple ECUs and systems can access information simultaneously, making the link full-duplex. The common Master/Slave method is used to determine the clock between systems.

2.3.3 Ethernet backbone network

Cost is a substantial incentive for the automotive industry, which is one of the reasons that there will never be only one communication technology used throughout a vehicle. For menial tasks, it is better to use a low-cost technology rather than a complex and more expensive technology. However, the interconnection possibilities of the ECUs would benefit from a switched backbone network. In that way, the different tasks in a vehicle will only employ the technology most suitable for its case and still be connected with other parts of the vehicle. The use case for Ethernet in automotives would be as the backbone network. Instead of ECUs having interfaces for several bus technologies, they would employ a specific one and then communicate to a gateway which in turn can encapsulate the message and forward it via Ethernet frames. It would allow any ECU to communicate via these gateways, where the Ethernet frame could add/strip headers to match the frames of e.g. CAN or FlexRay. The identifier concept used in broadcasting schemes, e.g. identifiers in CAN or assigned transmission slots in FlexRay, would be related to destination ports/addresses on a gateway [26].

Furthermore, a switched system is more fault tolerant than a wired bus where a single faulty node can break the entire communication. Fortunately, in a switched network a faulty node will only affect its direct neighbours. If one part of the backbone network is turned off, the remainder of the nodes can still operate [27].

There are many different areas in a vehicle that utilize and depend on communication technologies. However, the demands for each area vary significantly. Table 2.4 displays different domains in a vehicle and examples of their respective tasks [28]. Figure 2.7 represents a possible scenario of Ethernet implemented as a backbone network, where ECUs with technology best suited for specific domains are still employed and can further communicate via Ethernet. This would reduce the cost for individual ECUs as well as providing better bandwidth possibilities [26].

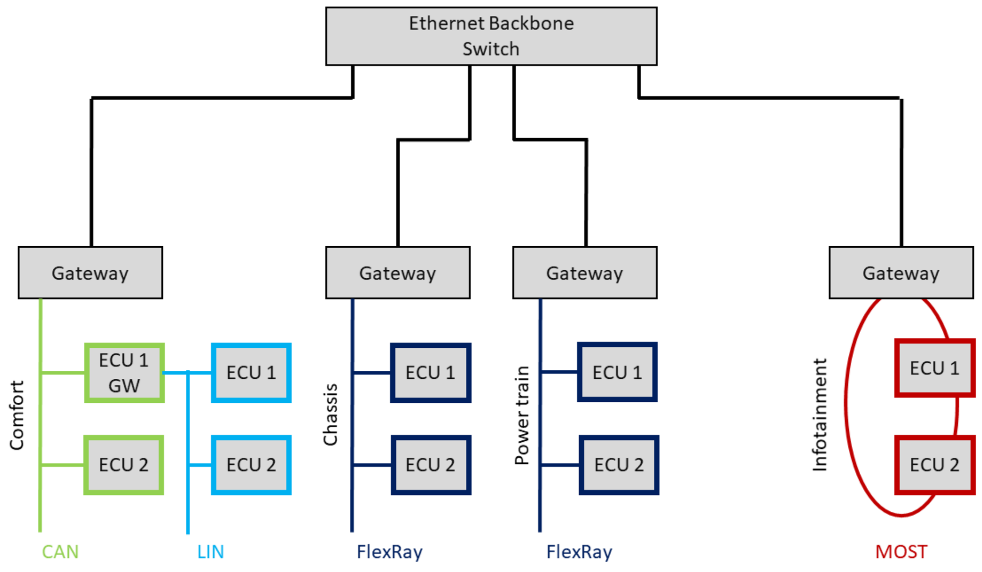


Figure 2.7: Representation of how a potential Ethernet backbone network could function in a vehicle.

Table 2.4: Vehicular functional domains and application examples [28].

Functional domain	**Applications**
Powertrain	Control data from e.g. engine, gearbox etc.
Chassis	Control data from suspension, steering, braking etc.
Body & Comfort	Driving unrelated data from e.g. climate control, mirrors, window lifts etc.
Driver assistance	Control data such as speed limit information, lane departure warnings etc.
Telematics/Infotainment	Presentation data from e.g. dashboard, head-up display etc.
Entertainment	Driving unrelated data from e.g. hand-free phones, rear seat entertainment etc.

2.3.4 Ethernet AVB / TSN

Audio Video Bridging (AVB) is an extension to Ethernet designed to provide time synchronized, deterministic and low latency streaming services. One of the reasons work began on developing AVB as an extension for Ethernet was due to the significant increase in Audio/Video applications and ECUs overall. In a vehicle these applications extend to camera devices and infotainment devices, but there is also a significant increase in control data. The extension was developed as a standard by IEEE and the further advancement on the AVB standard was renamed to Time-Sensitive Network (TSN). The development includes a set of IEEE standards from the 802.1 family and interacts on layer 2 in the OSI-model. The base standards can be seen in Table 2.5. In order to synchronize devices in the network, the Precision Time Protocol (PTP) along with the Best Master Clock Algorithm (BMCA) is used. The BMCA determines a grandmaster, i.e. a master node, from which the reference clock is sent to the slaves through PTP messages. An example of how PTP messages distribute clock values can be seen in Figure 2.8. This ensures a precise synchronization among nodes within the network [24], [29], [30].

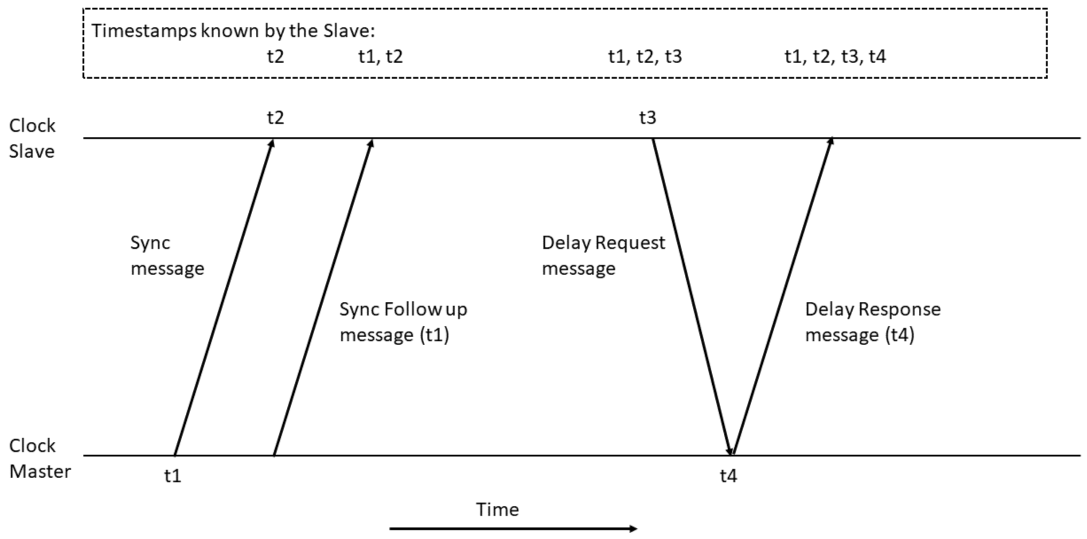


Figure 2.8: An example of how PTP messages can distribute clock values used to synchronize nodes in a network.

Table 2.5: The base standards of TSN [29].

Standard	**Title**
IEEE Std 802.1Q - 2018	Bridges and Bridged Networks
IEEE Std 802.1Qbv - 2016	Enhancements for Scheduled Traffic
IEEE Std 802.1AB - 2016	Station and Media Access Control Connectivity Discovery (specifies the Link Layer Discovery Protocol (LDPP))
IEEE Std 802.1AS - 2011	Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks
IEEE Std 802.1Qav - 2009	Forwarding and Queuing Enhancements for Time-Sensitive Streams
IEEE Std 802.1AX - 2014	Link Aggregation
IEEE Std 802.1BA - 2011	Audio Video Bridging (AVB) Systems
IEEE Std 802.1CB - 2017	Frame Replication and Elimination for Reliability

2.3.4.1 System Configuration

When implementing a TSN network there is much configuration needed, to allow for a stable flow between two endpoints (talker/listener) via bridges (Ethernet switches). A centralized approach is to use two logical entities, with interfaces to the endpoints and bridges, called the Centralized Network Configuration (CNC) and the Centralized User Configuration (CUC). The CNC has global knowledge of network resources and topology. It manages the connections in the network and acts as a proxy for the network. Requirements for connections is provided by the CUC, which establishes the requirements with the endpoints. Knowing the communication requirements, the CNC can organize the transmission paths for the streams between the talkers and listeners. A representation of the communication flow of the centralized configuration approach can be seen in Figure 2.9. There is potential for this configuration to be automated in the future development of TSN [24], [31], [32].

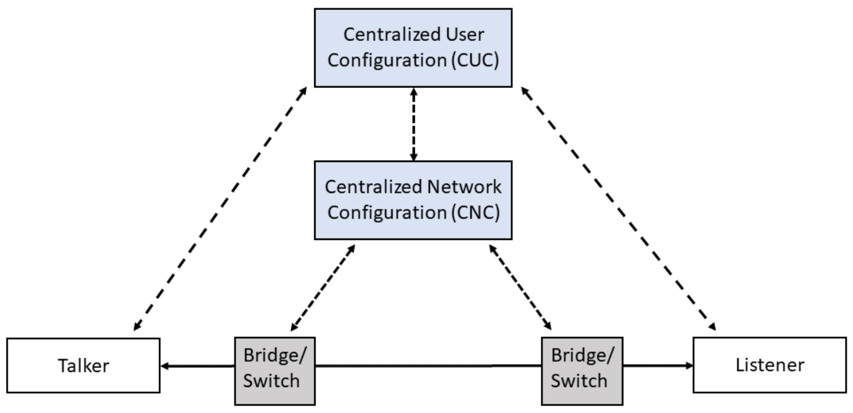

Centralized User Configuration (CUC)
Centralized Network Configuration (CNC)
Talker
Bridge/ Switch
Bridge/ Switch
Listener

Figure 2.9: A representation of a centralized configuration setup. The endpoints communicate requirements to the CUC which informs the CNC, which then can organize transmission path for the streams of the network.

2.3.5 System Description

A problem with Ethernet in automotives is that it is not suitable for real-time and safety-critical applications. This is why several extensions to Ethernet is under development and of great interest for the autonomous drive industry. In 2005, the so-called Best Effort (BE) traffic was introduced. This allowed prioritized traffic to have a higher Quality-of-Service (QoS). Following BE, the AVB standard was developed and later turned into the TSN task group with focus on safety-critical and time-sensitive transmissions. TSN employs the Stream Reserve Protocol (SRP) to establish AVB streams. Furthermore, TSN divides AVB traffic into two types: stream reservation class A and B (SR-A & SR-B). They differ in terms of maximum allowed latency, for SR-A a latency of 2 ms is required and for SR-B a latency of 50 ms, over seven hops. All the legacy Ethernet frames are covered by the Best Effort class, which employs a priority based scheduler. The concept of Time-Triggered (TT) traffic was also introduced. For this extended Ethernet, there are now three traffic types available: BE, AVB and TT. They are prioritized differently, which makes them suitable for different applications. TT the highest priority and BE the lowest, AVB class A has higher priority than B. In automotive Ethernet networks these three traffic types will be present, resulting in a complex transmission schedule. There is a specification that ensures that class-A AVB and TT traffic can only reserve 75% of the total bandwidth, this is to prevent starvation of the less prioritized traffic [33], [34].

The egress ports of the bridges has a prioritized queue, ranging from $0-7$ where 7 is the highest priority. Every frame contains a priority field that matches the bridges queue. The TT traffic typically has the highest priority, followed by the two classes of AVB streams and the five remaining queues for different BE traffic. The transmission schedule employs so-called Transmission Selection Algorithms (TSA). For AVB queues the most common is the Credit-Based Shaper (CBS) and for the TT queue, the Time Aware Shaper (TAS). An example of a typical bridge configuration, containing these elements, can be seen in Figure 2.10. The gates for each queue is further controlled by a TAS, which Figure 2.11 is an example of, that determines whether the port-specific gates should be opened or closed. Briefly explained, TAS organizes the traffic into periodic cycles where TT traffic is prioritized. There is a time reference distributed between the bridges and as the example in Figure 2.11 shows, the entry T000:10000000 in the control list implies that at the relative time "T000" the highest priority queue is open (1) [27], [34].

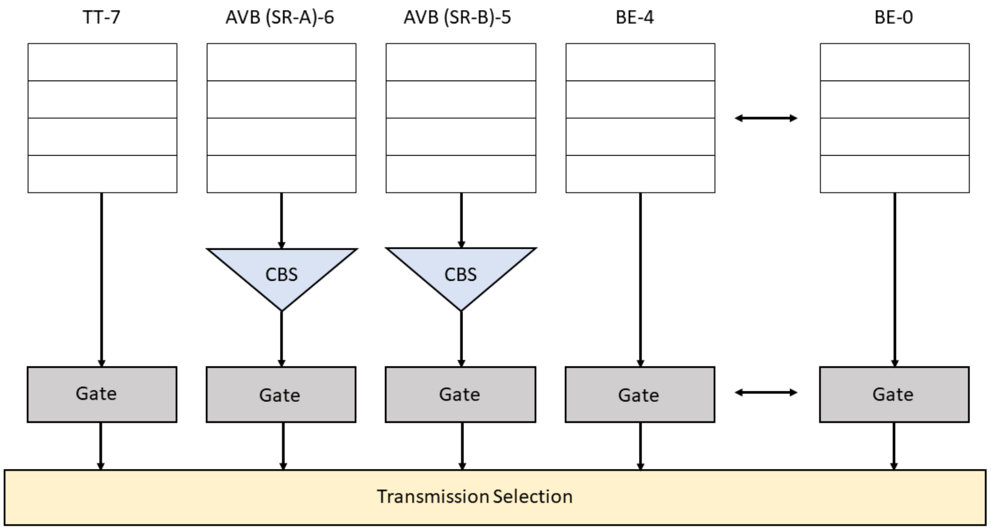


Figure 2.10: A representation of how a typical TSN bridge is configured. The port-specific gates are controlled by a TAS, see Figure 2.11 for an example.

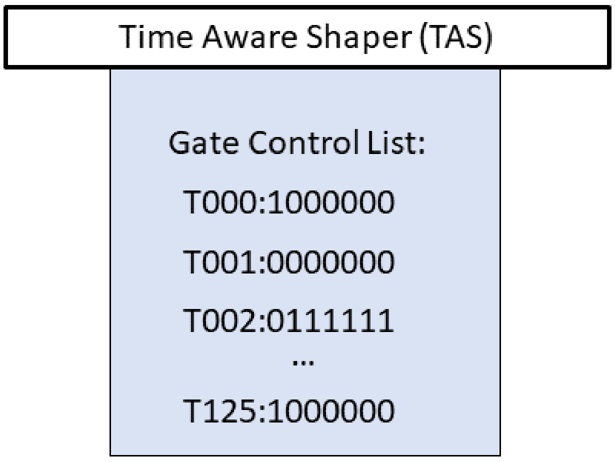


Figure 2.11: An example of a TAS, where at relative time "T000" only the highest priority traffic is allowed (i.e. TT traffic).

AVB frames are only allowed transmission if: (i) the AVB queue is open, (ii) there is no higher priority frame being transmitted, (iii) if the CBS allows it, i.e. if the available credit count is greater than or equal to zero. An example of a transmission schedule is presented in Figure 2.12. The credits are initialized to zero and increases over time when no AVB transmission is present, with a configuration parameter called *idle slope*. Credits are decreased, with a so-called *send slope*, when AVB frames are being transmitted and they are frozen if the gate is closed. If the AVB queue is emptied while positive credit, it is reset to zero. It is when the credits are negative that the BE traffic potentially is allowed transmission. Given that the gates are open for the BE queues and that no higher prioritized frame is being transmitted. During BE traffic transmission the credits are regenerated and if they reach a non-negative value, pending AVB frames will be transmitted instead [33], [34], [35].

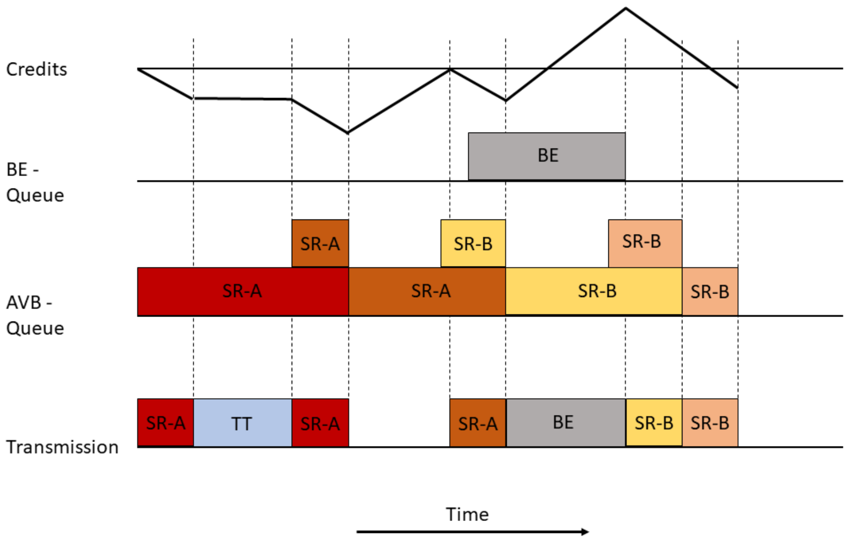


Figure 2.12: An example of how the transmission is scheduled in case of TT, AVB and BE traffic. The credit development is included, with the *idle slope* and *send slope* present. Worth noticing is also how the credits are frozen during a TT transmission and that due to the credits being negative, the next AVB frame is not allowed transmission.

2.3.5.1 Frame Format

As an extension to the regular Ethernet frame, TSN includes an addition to the VLAN Tag segment. A representation of an Ethernet frame with the TSN addition is displayed in Figure 2.13. The frame consists of a preamble and a Start of Frame pattern (SOF), which are then followed by the layer 2 MAC destination and source addresses. Following this is the IEEE 802.1Q VLAN Tag segment, which consists of four fields: Tag Protocol Identifier (TPID), Priority Code Point (PCP), Drop Eligible Indicator (DEI) and a VLAN Identifier (VID). The PCP indicates the priority of the data, i.e. what traffic class the frame belongs to. The DEI is a congestion protection, as it indicates if the frame can be dropped in case of congestion, and the VID specifies which VLAN the frame belongs to. Also included in a frame is the Ethertype (or length for small frames), which specifies what protocol (if any) is transported in the frame. The main contribution in a frame is the payload, which is followed by a Frame Check Sequence (FCS); a 32 bit CRC. Furthermore, in between two Ethernet frames there is minimum inter-frame gap (IFG) [32], [35].

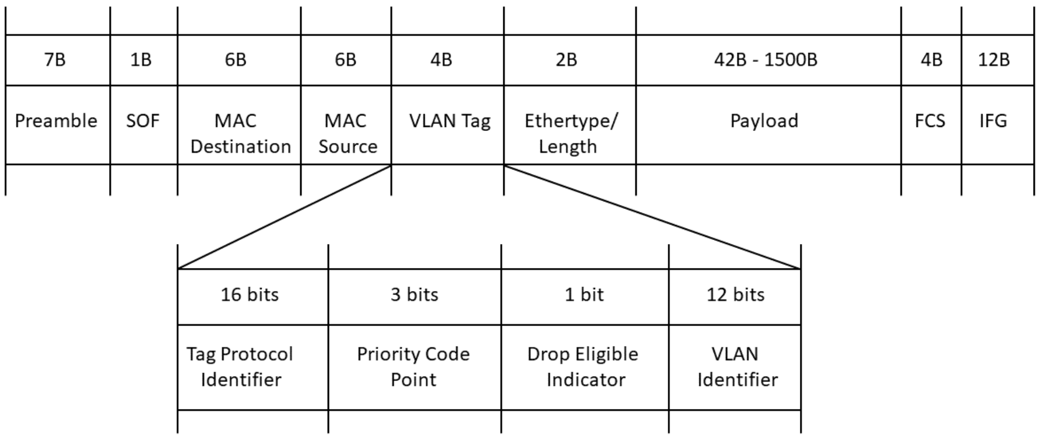


Figure 2.13: Representation of an Ethernet frame, with the TSN extension to the VLAN Tag. Included is also the minimum inter frame gap (IFG) between two Ethernet frames.

2.4 AUTOSAR

Traditionally in the automotive industry, the way to develop new electrics and electronics (E/E) has been to have one unit for every service. In modern vehicles, the architecture connecting the E/E area has increased significantly in complexity. The AUTomotive Open System ARchitecture (AUTOSAR) was launched in order to provide a standardized architecture to the basic software and interfaces to applications. A standardized architecture allows for compatibility between different Original Equipment Managers (OEMs) and sub manufacturers. One can imagine the software and hardware as being two puzzle-pieces interconnected in a very specific way for each OEM. However, the AUTOSAR architecture functions as a middle ground which allows any software to communicate with any hardware and an illustration can be seen in Figure 2.14. This abstraction of software from hardware is the core function of the AUTOSAR architecture and it allows for more flexible development in the automotive industry. Another important aspect of AUTOSAR is to allow reusability of functions across vehicle networks and OEM boundaries [2].

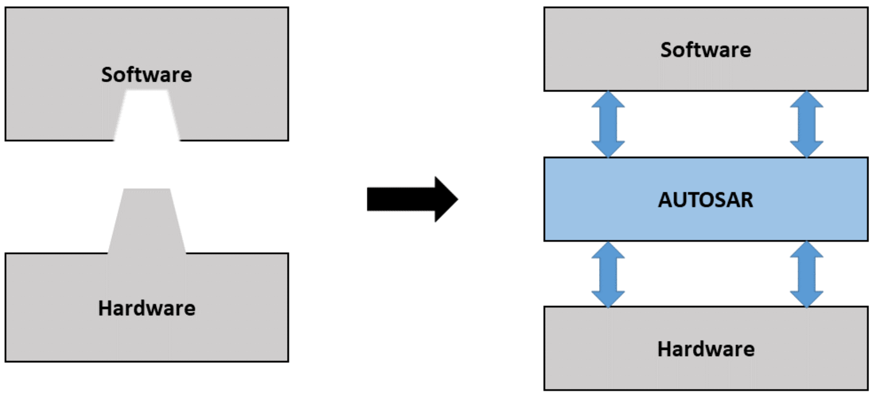


Figure 2.14: Illustration of how the AUTOSAR architecture relieves the constraint of matching specific software to hardware, by abstracting the two from each other.

2.4.1 AUTOSAR Infrastructure

The AUTOSAR architecture consists of a layered topology, which includes three main layers that each has a specific purpose. The layered architecture provides a level of abstraction between the different layers, resulting in the middleware solution that is adaptable to any OEM solution regarding both software and hardware. A drawback is that the layered architecture requires more available memory and computing power, as you need the entire AUTOSAR stack for any application. This can also make a simple software task become complicated since all implementations must follow the AUTOSAR methodology.

2.4.1.1 Basic Software

The bottom layer, i.e. the layer connected to the hardware, in the architecture is the Basic Software layer (BSW). This layer itself consists of several sublayers containing different modules which are used by the Application Layer via the RunTime Environment (RTE), such as events and timers. The BSW contains all necessary modules in order to perform a complete abstraction of Software components (SWC) and it is often standardized by AUTOSAR. If a module is missing to a specific SWC, one may implement a Complex Driver. It allows users to implement standalone extensions to an AUTOSAR application, or e.g. enhance legacy functions. None of which are however supported by AUTOSAR [36], [37].

2.4.1.2 Runtime Environment

The RTE, also known as Runtime Infrastructure, is a middleware that abstracts the network topology for inter- and intra-ECU information exchange between the application SWCs and also between the BSW and applications. In other words, the RTE maps and configures the SWC runnables to OS tasks and lets events trigger the runnable SWC. This is performed by a Virtual Function Bus (VFB) and acts as the communication medium between SWCs and the BSW. The RTE itself does not contain any runtime components and functions more as a barrier, concatenating functionality between the SWC and the OS/BSW without letting the two sides in contact.

2.4.1.3 Application layer

The Application Layer contains the collection of SWC applications that interact with the RTE. The application layer also uses the VFB to communicate with other SWCs, either in the same ECU or in the network, by defining a connection of connectors and ports. These connections may be of two types; Sender-Receiver and Client-Server. How these may be implemented is specified in the RTE and their usage in the VFB [38].

2.5 SOME/IP

The classic bus systems in automotives are based on a signal-oriented approach, where the sender decides whether data should be transmitted or not independent of a receivers request. Consequently, a significant amount of non-requested data is occupying the bandwidth which may cause internal communication problems. SOME/IP is a middleware using a different approach, namely service-oriented communication. In contrast to signal-oriented systems, service-oriented systems communicate data exclusively on the receivers demand and not when a sender finds a transmission appropriate, hence an increase in qualitative bit rate. An illustration of the difference between the two orientations can be seen in Figure 2.15. Two important concepts in the architecture are the Service Discovery (SD) and Publish/Subscribe (Pub/Sub). SD allows each node (ECU) to dynamically find a variety of functionality among other nodes and configuring access, i.e. set up a subscription to a node which is illustrated in Figure 2.16. As a result, a newly connected node may find functions without being preconfigured with the knowledge of which nodes that hold the specific functions. Nodes may also Pub/Sub, allowing nodes to decide which content to communicate in an active subscription. Nodes may also use other nodes' functionality and methods by a remote procedure call (RPC). However, all nodes may not operate on the same OS and the communication might become incomprehensive. In order to solve this particular problem, SOME/IP implements serialization which parses structures such as RPC Protocol Data Units (PDU) in AUTOSAR and converts these to byte streams ready for transmission. When a node receives the byte stream, the SOME/IP protocol performs a deserialization to convert back to the appropriate structure for the specified OS [3], [4].

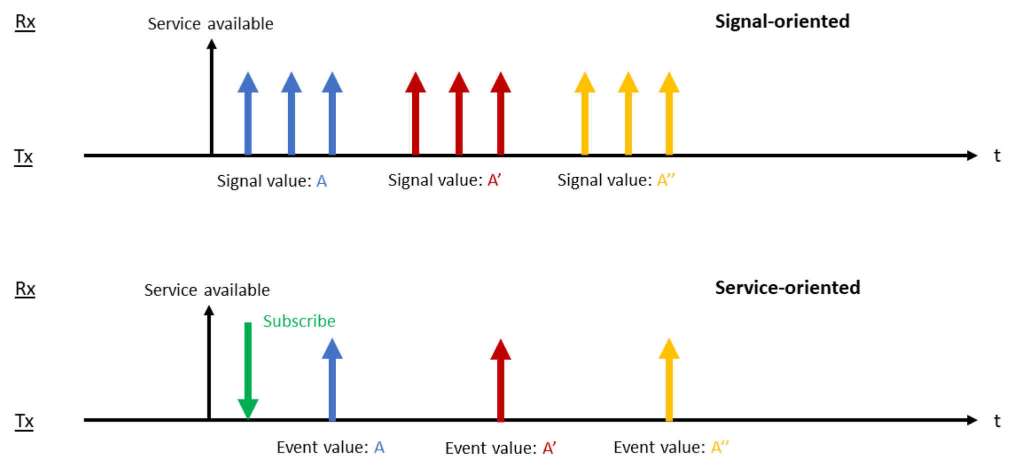


Figure 2.15: Illustration of a bus system with a signal- and service-oriented approach. Worth noting is how in a signal-oriented approach the bus is occupied by a continuous transmission of signal values.

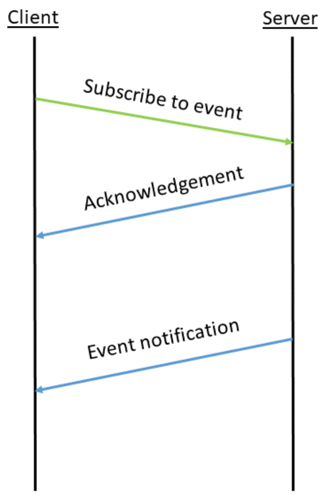


Figure 2.16: Illustration of the service-oriented approach, used in a some/ip system, with a client/server methodology of subscribing to an event.

2.5.1 Header format

As SOME/IP is based in the higher layers of the OSI-model, it only contributes with a header to the data frame. Further embedded into the frame will be lower layer headers, e.g. TCP/IP headers on an Ethernet frame. An example of a SOME/IP frame is displayed in Figure 2.17. Highlighted is the SOME/IP header format, where the Message-ID field consists of a 16 bit Service ID and a 16 bit Method ID. The former is used to identify specific services, as each service must have a unique ID. A service can consist of several methods, events and fields which is what the Method ID is a reference to. The concept of the Message-ID is similar to the CAN ID, which allows for enhancing/adopting these to a SOME/IP structure. The length field simply specifies the number of bytes in the payload, some header information and the Request/Client ID, and is assigned 32 bits. The Request ID field contains a 16 bit Client ID and a 16 bit Session ID. The purpose of this field is to differentiate multiple calls of the same method. The Client ID allows for identifying specific clients, whilst the Session ID differentiates the multiple calls from each specific client.

Furthermore, there are four 8-bit segments before the payload (which is of variable size as it contains headers and payload from lower OSI-layers). The Protocol Version field simply contains information about what SOME/IP protocol version is used and the following field is the Interface Version field, which defines the service interface version. There is also the Message Type field, which differentiates between the possible types of SOME/IP messages shown in Table 2.6. The last field is the Return Code, which indicates if a request was successfully processed or not [13], [39].

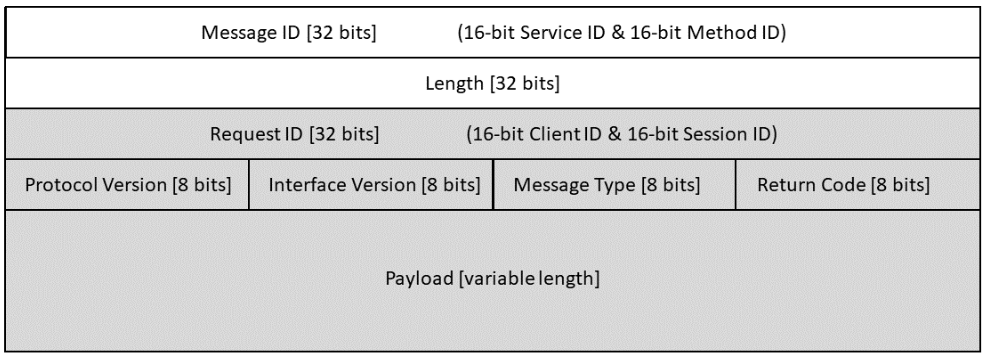

Message ID [32 bits]
(16-bit Service ID & 16-bit Method ID)
Length [32 bits]
Request ID [32 bits]
(16-bit Client ID & 16-bit Session ID)
Protocol Version [8 bits]
Interface Version [8 bits]
Message Type [8 bits]
Return Code [8 bits]
Payload [variable length]

Figure 2.17: The header format of a SOME/IP frame, where the payload is of variable length as it depends on what lower layer headers that are present. The highlighted areas in gray is covered by the value in the Length field.

Table 2.6: Possible types of a SOME/IP message.

Number	**Value**	**Description**
$0x00$	REQUEST	A request expecting a response (even void).
$0x01$	REQUEST_NO_RETURN	A fire & forget request.
$0x02$	NOTIFICATION	A request for a notification (i.e. a subscription to an event call back or a field value), expecting no response.
$0x80$	RESPONSE	The response message.
$0x81$	ERROR	In case a response message cannot be delivered due to an error.
$0x20$	TP_REQUEST	A TP request expecting a response (even void).
$0x21$	TP_REQUEST_NO_RETURN	A TP fire & forget request.
$0x22$	TP_NOTIFICATION	A TP request for a notification (i.e. a subscription to an event call back or a field value), expecting no response.
$0x23$	TP_RESPONSE	The TP response message.
$0x24$	TP_ERROR	In case a TP response message cannot be delivered due to an error.

2.5.2 SOME/IP-SD

A single ECU can contain multiple clients and services, there can also be several instances of the same service on different ECUs. The term *client* refers to a node that is requesting a subscription to a specific service. In SOME/IP terminology, a client can either be active or down and the same goes for a service. In the active mode, there are three functioning phases that a client and service can take: the *initial wait phase*, the *repetition phase* and the *main phase*.

The SOME/IP Service Discovery (SD) functions by services broadcasting so-called *offer messages* on the network. It is upon receiving such a message that a client, if interested, can subscribe to the service. To reduce subscription time, clients can also broadcast messages of their own to request a specific service. These are called *find messages* and to which a client may receive offer messages in response. Another important aspect of the SOME/IP-SD is that a random waiting delay is implemented in the sequence of operations. This ensures that not all services and clients start to operate at the same time, which in turn relieves some network and CPU load. Upon a completed subscription, a subscriber may receive two notification formats: Event and Field Notification. Event Notifications provide data for the specific event and acts as a Fire & Forget, whereas Field Notifications contain data related to previous events and are therefore often equipped with set and get-methods [3], [40].

2.5.2.1 *Initial Wait Phase*

A client in the initial wait phase is initialized to this phase on the request from the application layer and remains in it for a time randomly chosen according to SOME/IP specifications. Whilst in this phase, the client remains silent. Only when an offer of the specific service that the client is interested of, it takes action. It then subscribes to the service and enters its main phase.

Similar to a client, a service enters the initial wait phase when it is set to available by the application layer. It remains in this phase for a time randomly chosen and it does not send any messages during this time. However, unlike a client, any received find message will be ignored and a service cannot skip the repetition phase.

2.5.2.2 *Repetition Phase*

In the repetition phase, a client will send a predefined number of find messages (according to specifications) with an exponentially increasing waiting time between the successive messages. Again, if a client receives an offer message from a service it is interested in, it will request a subscription to this service and move to the main phase.

Upon entry to the repetition phase, a service broadcasts an offer message. Similarly, the successive messages are sent with an exponentially increasing delay. If a find message from a client is received, the service waits for a chosen random time before responding with a unicast offer message. A service moves from the repetition phase to its main phase after a predefined number of offer messages sent.

2.5.2.3 *Main Phase*

A client in the main phase will not send any find messages on its own, only if a server's offer message must be answered. However, a service will cyclically send offer messages as well as answer find messages.

2.5.2.4 Functioning Modes

Besides being active/down and in different phases, a client can be configured in two different modes: the *request* mode or the *listen* mode. In the former, a client will send find messages in the repetition phase whereas in the latter it will only wait for offer messages. A service also has two configurable modes: the *offer* mode and the *silent* mode. In which it will send offer messages in the repetition and main phase, or only respond to find messages [3], [40].

2.6 Adaptive AUTOSAR

In the classic AUTOSAR system, it can be quite difficult to update and upgrade the individual ECUs. This means a lack of defense against new security risk as well as difficulties when integrating new functions. As this has become more sought after properties, to be able to dynamically reload software components and perform more computing-intensive tasks with large data sets, a new POSIX based standard called the Adaptive AUTOSAR Platform has been developed. It employs a more object-oriented approach which allows for more flexible options in the vehicle ECU architecture. It alleviates application development by being able to upgrade any ECU instead of having to perform individual development for each ECU. Applications can also be integrated into the system at runtime due to it building on service-oriented communication. The main use cases thought for the new AUTOSAR standard are more algorithms for automated driving and better possibilities for V2X communication as well as multimedia applications [41], [42].

The Adaptive AUTOSAR Platform is thought to supplement the classic AUTOSAR standard. An approach which allows for more dynamic software configurations and development. With the service-oriented communication concept and heterogeneous computation, it provides better performance as well as compatibility with e.g. SOME/IP from the classic platform.

2.6.1 Technology Drivers

What has been the main technology drivers behind the work on a new platform, are Ethernet and processors. As Ethernet has been introduced to the automotive industry, offering a higher bandwidth and with switched networks enabling a more efficient communication, the classic platform is designed for the legacy technologies and it is optimized for such. It is therefore difficult to fully benefit from an integrated Ethernet backbone network on the classic platform.

As the bandwidth demands are growing, similarly are the performance requirements for processors. There is already multicore processors in use with the classic platform, but multicore might not be enough in the future. It is highly likely that so-called Manycore processors, with tens to hundreds of cores, will be needed. This increase in number of cores would overwhelm the design of the classic platform, which was only developed for a single core [41].

3
Implementation of a Communication Link

In this chapter, the focus lie on the procedure of developing a practical model of a communication system. The model will function as an evaluation tool for performance metrics of an AUTOSAR node interconnected with a small Ethernet network. Furthermore, sections regarding the different development environments and software architectures used will be described as well.

3.1 Development Environment

The hardware used was provided by QRTECH in the form of an ECU product called QRx. Two of these were used, one running on AUTOSAR OS and the other on Linux OS. The platform implementation and source code for the AUTOSAR node was developed by the Swedish software company ArcCore [43].

Arctic Studio

The software platform that was used when programming the AUTOSAR QRx is called Arctic Studio. It is a rebranded Eclipse release with modifications applied to comply with AUTOSAR development and is also a product from ArcCore. The coding language used was C++.

Visual Studio

For some of the simpler programs used, such as a simple TCP chat program, the software was written in the language C# in the Microsoft Visual Studio platform. Most of these programs were used as the initial setup when developing the system, to allow testing the different parts of the system with ease.

Linux

In order to build and run programs on the Linux system, cross-compiling instructions for an ARM v.7 Cortex-architecture were required. Moreover, a GDB server was used to remotely debug the devices from a computer via an SSH connection. A GDB server is a GNU Debugger and is used to control the execution of a program and lookup variables between loops etc. The coding language used was C and the platform was Eclipse.

3.2 System model

The model involved the setup of a communication link between two ECUs running on different operating systems. The communication stack implemented was TCP/IP which further utilizes encapsulation in Ethernet frames. Included in the system was also two PCs, running Windows 10, that were used as control and presentation devices. A representation of the system model can be seen in Figure 3.1. The model was constructed to mimic a real scenario within a vehicle, where one ECU running on AUTOSAR has less computational power and only transmits data to a more powerful ECU running Linux. The TCP/IP stack was implemented in both OS, including the necessary socket setups as described in Section 2.1.3. This allows for TCP packages to traverse the Ethernet cable and be accepted on the receiving end.

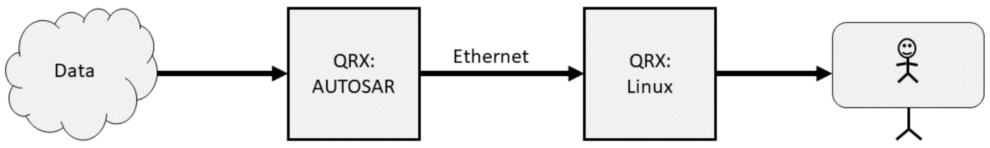


Figure 3.1: Representation of the system model that was constructed. It aims at mimicking an automotive scenario where a more central located computational prominent ECU (Linux) gets fed data via a computationally weaker endpoint ECU (AUTOSAR). The connection between the ECUs is an Ethernet cable and the sink of the model is a simple presentation stage.

3.2.1 Source

The initial part of the system model is the source of data, e.g. an image feed from a camera. The acquisition of data was not the center of attention, as it is the actual transmission over the Ethernet network that was of interest. The data could e.g. be an image feed from a sensor which should be forwarded to a neural network which then can perform decisions based on the data. For simplicity, however, the data feed was implemented in a C# program in Visual Studio that initialized a socket connection with the AUTOSAR node and to which it forwarded images stored on the PC. The C# program utilized the Console Application functionalities enabling it to load images and using the socket send function to transmit the raw bytes. A 4-byte header was added to the images which indicate the number of bytes that each image represents. As the communication is a steady flow of data, the 4-byte header allowed the sink to be able to reconstruct each image.

3.2.2 AUTOSAR node

The source of data was transmitted to the AUTOSAR node in the link, which purpose was to forward the data with high speed and with a reliable transmission. No real processing is done on the data, it is only temporarily stored and then forwarded to the Linux end. The 4-byte header was treated as regular data and forwarded along with the rest of the image bytes.

The implementation on the AUTOSAR node included setting up two server sockets on initialization of the ECU. With connection on both of these sockets, communication over the link could be initiated. The TCP/IP module for AUTOSAR was implemented with interrupts that trigger when a listening socket has received a packet. These interrupt functions were utilized to fill one out of two buffers which were switched between a forwarding segment and the interrupt section. By always having one buffer being filled and one being emptied, the AUTOSAR node should have a good forwarding throughput.

The TCP/IP stack implemented in the AUTOSAR configuration was the lightweight IP (lwIP) which is an open source TCP/IP stack designed for embedded systems to reduce RAM usage. Due to this, a code extension to the TCP/IP stack had to be implemented as the initial tests showed a flaw in the configuration. Sockets were accepting packets even though the data never had reached the application layer. A small segment, referred to as TCP Throttling, was therefore added, which purposely decreases the source's transmission speed if the receiver cannot keep up with processing the received data. Practically, the method will block the TCP module on the AUTOSAR side until the received content has been forwarded to the application. Hence, AUTOSAR will not reply to the source with an ACK and therefore all TCP communication is paused during the data processing duration. Upon data process completion, a boolean flag in AUTOSAR is set to activate TCP again meaning that the node has processed all data and is ready for new content.

Another issue concerning the TCP/IP implementation was the transmission from the AUTOSAR node. The system would not correctly check if a packet had been successfully sent to the next node. As a result, the data experiencing a failed transmission would not be retransmitted. The packet was instead marked as sent, causing loss of data over the link. Therefore, another code segment was implemented, referred to as TCP Transmission Check, to check if all data had been successfully transmitted. The implementation is based on a while-loop in the forwarding segment. The original problem is related to the TCP transmitter buffer queuing up and being filled, resulting in errors when attempting to send new packets. If this error occurs, the loop continues until all data is successfully accepted into the buffer and transmitted.

3.2.3 Linux node

The Linux node is similar to the AUTOSAR side of the link in terms of functionality, as its main purpose was to receive and forward data to another node in the link. In addition, the Linux node also interprets the first 4-bytes of each image for control and debugging purposes. In the thought-out automotive scenario, the Linux system would perform a number of complex calculations on the data, considered too demanding for AUTOSAR, and then either respond or forward the processed data further into the network.

The Linux system first establishes two TCP client sockets that are connected to the AUTOSAR and the Sink. The application program uses a state machine methodology, which upon a successful connection is initialized to enter the first out of three states: RECEIVE, STREAM and STOP. In the RECEIVE state, the system simply awaits incoming data. Upon arrival, the first 4-bytes are retrieved and stored in the message buffer as well as transitioning to the state STREAM.

The STREAM state consists of loops which analyze the buffer status, whether it is full or close to maximum capacity, and perform actions based on the result. Briefly presented, the system receives data until the whole image is received or the buffer has less than one TCP packet payload in storage. On these events, the system will forward the stored data and reset buffer indices. When the whole image is received, the state STOP is set and the last segment of data will be forwarded. Following is a reset of buffers and the state transitioning back to RECEIVE. The system is now ready to receive the next image and to repeat the whole process.

3.2.4 Sink

In the final step of the link, the Sink of the system is a simple presentation of the data received from the Linux node. The implementation was performed similarly to the Source by utilizing a Windows Form Application in Visual Studio and TCP sockets. The received data represented raw bytes of an image which was converted to a bitmap, which in turn allowed it to be portrayed in a so-called picture box extension to the Windows Form. Like the Linux node, the Sink interprets the 4-byte header to be able to reconstruct each image from a continuous flow of data.

4 Results and Analysis of the Link Performance

This chapter presents the results obtained during the evaluation of the constructed communication link as well as an analysis of the performance. The results are then further discussed in Chapter 5.

4.1 The Communication Link

The constructed model described in Section 3.2 was able to establish communication between two ECUs utilizing TCP/IP and Ethernet. The model aimed at mimicking an automotive scenario where an ECU with less computational power, running AUTOSAR, is able to transmit its vital data to a more powerful ECU, running Linux, via Ethernet frames. The link was evaluated with the help of the program Wireshark, which allows for network sniffing and packet capturing.

Even though there are well-defined standards for established protocol stacks, there is always some work included when implementation them into a system. The implementation of the TCP/IP stack on the AUTOSAR node is not a one to one mapping with the AUTOSAR standard. A few simplifications and implementation choices have been made by the developer of the platform, such as implementing a lwIP stack in the TCP/IP module. This resulted in performance complications.

4.1.1 TCP Throttling

With the added throttling implementation to the TCP/IP stack, mentioned in Section 3.2.2, the initial transmission results were improved and the stack behaved more normal. When running a simulation on the SW without the throttling addition it could be seen how on socket level, the transmissions kept on being accepted even though the data was never forwarded to the application. With the addition of the throttling, however, this behaviour stopped and the correct size of data was being forwarded.

4.1.2 TCP Transmission Check

The addition of the transmission check segment, mentioned in Section 3.2.2, relieved the issue of a mismatch in data received compared to data transmitted. In the case of transmission errors, the application SW did no longer treat that data segment as a successful transmission but instead as data needed to be retransmitted. However, this did not solve the reason as to why the transmission errors occurred in the first place. Due to this error the code would never return from the loop, which resulted in large transmission delays that sometimes lead to timeout crashes. The transmission errors will be further highlighted in Section 4.3.

4.2 Round Trip Time

The round trip time (RTT) of a communication system measures the time it takes for a packet to be transmitted and the corresponding acknowledgement to be received. As a metric, it yields information about the propagation delay and interference on the link. It also indicates the potential for how many bytes in flight the link can exhibit.

A common way to measure the RTT is to analyze the initial TCP handshake, i.e. the time that transpires between the SYN, SYN/ACK and ACK. Wireshark provides this measurement, referred to as the initial round trip time (iRTT) and the measurements of this are shown in Table 4.1. The reason why RTT is commonly measured on the handshake is that it yields information about the base latency and the packets are very small, meaning they have a higher chance of traversing the link at higher speeds. Another advantage is that these packets are handled only by the TCP stack, i.e. there is no application interface involved which could be a source of delay. Worth noting is also that RTTs throughout the transmissions vary from one another as well as the iRTT, including both faster and slower times. Therefore, due to the retransmissions etc. that the link is experiencing, the iRTT provides a more stable result.

Analyzing the measurements displayed in Table 4.1, they appear reasonable with latencies of 1 ms or lower which should be normal for a small enclosed system. It is interesting that the communication between the AUTOSAR node and a PC experienced more latency than the other cases. However, a RTT of 1 ms does not indicate any prominent interference or propagation issues.

Table 4.1: iRTT Measurements

Endpoints	**iRTT (Average)**
PC $\longleftrightarrow$ AUTOSAR	$0,0010008$ s
AUTOSAR $\longleftrightarrow$ Linux	$0,0005693$ s
Linux $\longleftrightarrow$ PC	$0,0004455$ s

4.3 Data Throughput

The performance metric most centric to the evaluation of the link was the measurement of data throughput, i.e. the quantity of data flow and at what transmission rate. Wireshark has a feature that shows the throughput along with the length of individual packets (as blue dots) for each TCP stream. There is also a graph that shows the bit rate over transmission time, which should correlate somewhat to the throughput graph (slightly differ due to small differences in the capture process). These features were used to assess the Linux and AUTOSAR side on their own, as well as the full link.

4.3.1 Linux

The Linux system proved to be reliable, by achieving data rates close to the maximum capacity of 100 Mbps. A 202834 kB file was sent from the Source and received by the Linux node in approximately 20 seconds, meaning that the approximate bit rate was calculated as $(202834/20) * 8 = 81134$ kbps. The throughput graph of this can be seen in Figure 4.1, which shows how the data flow stays at a stable level. The result presented in Figure 4.2 indicates the same behaviour.

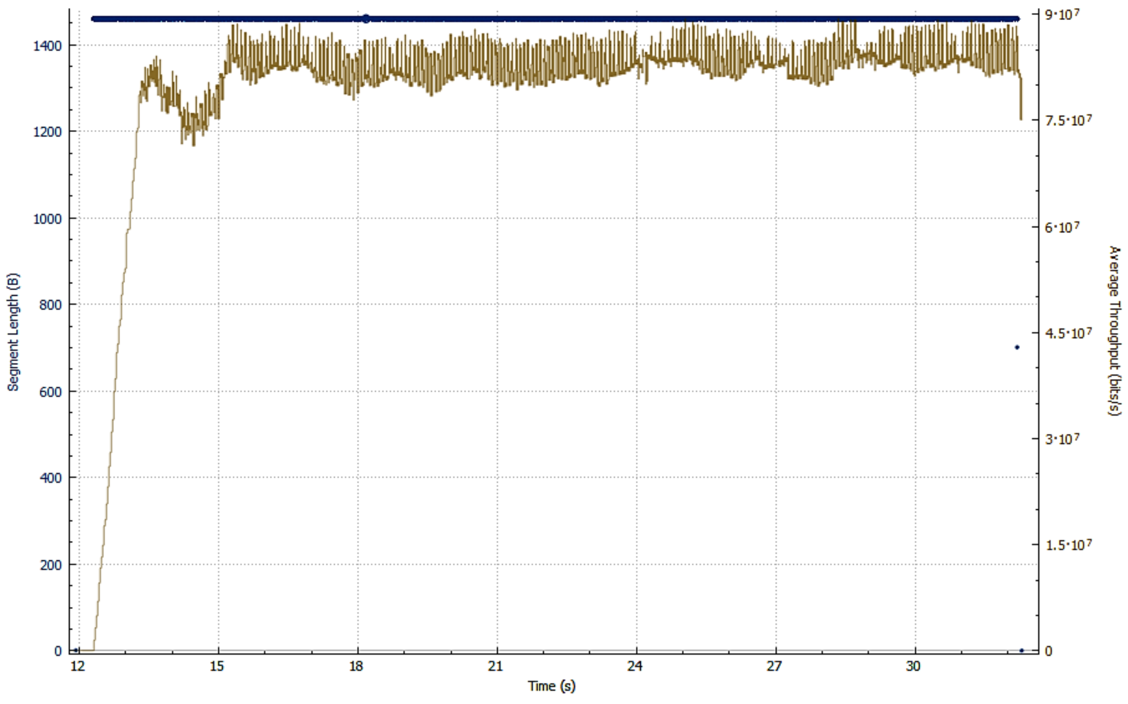


Figure 4.1: Wireshark throughput capture of the Linux node receiving a 202834 kB file from the Source, where the first 12 seconds is a startup phase of the system and not a part of the actual transmission. The average throughput level is stable and with a good average rate of around 80 Mbps

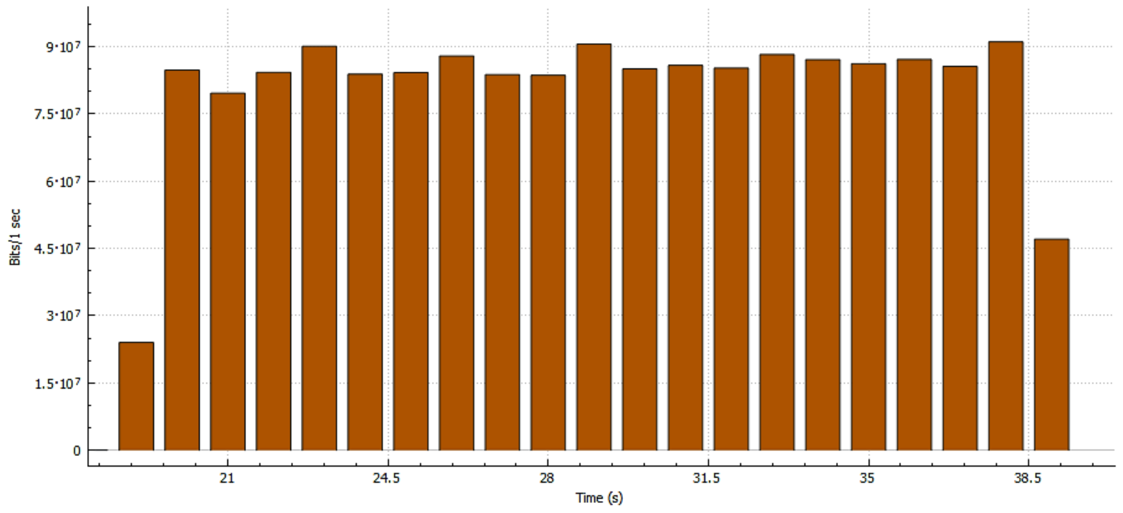


Figure 4.2: Wireshark IO graph of the Linux node receiving a 202834 kB file from the Source. It highlights the data rate over the approximate 20 seconds of transmission, where the average rate is around 80 Mbps.

The transmitter side is able to send the 202834 kB file to the Sink in 20 seconds as well, i.e. an approximate bit rate of around 80 Mbps, which correlates with Figure 4.3 and Figure 4.4. Just like in the receiver part, the data flow holds a stable level. However, there are now some packets with a length smaller than the MTU size of 1460 bytes. As can be seen in Figure 4.3, there are packets with payload lengths of 1428 and 32 bytes. Further inspection of these packets showed that every transmission with length 32 included a PSH (push) flag along with the ACK. The PSH flag is used to tell the receiving end to provide the read data segment to the reading application immediately (and stop any potential buffering). It should not have any negative effects on the result and the behaviour could be linked to TCP delay efficiency concepts such as the Nagle algorithm and delayed acknowledgments (mentioned in Section 2.1.4) or perhaps as a smart way of preventing the receiving window to be filled up.

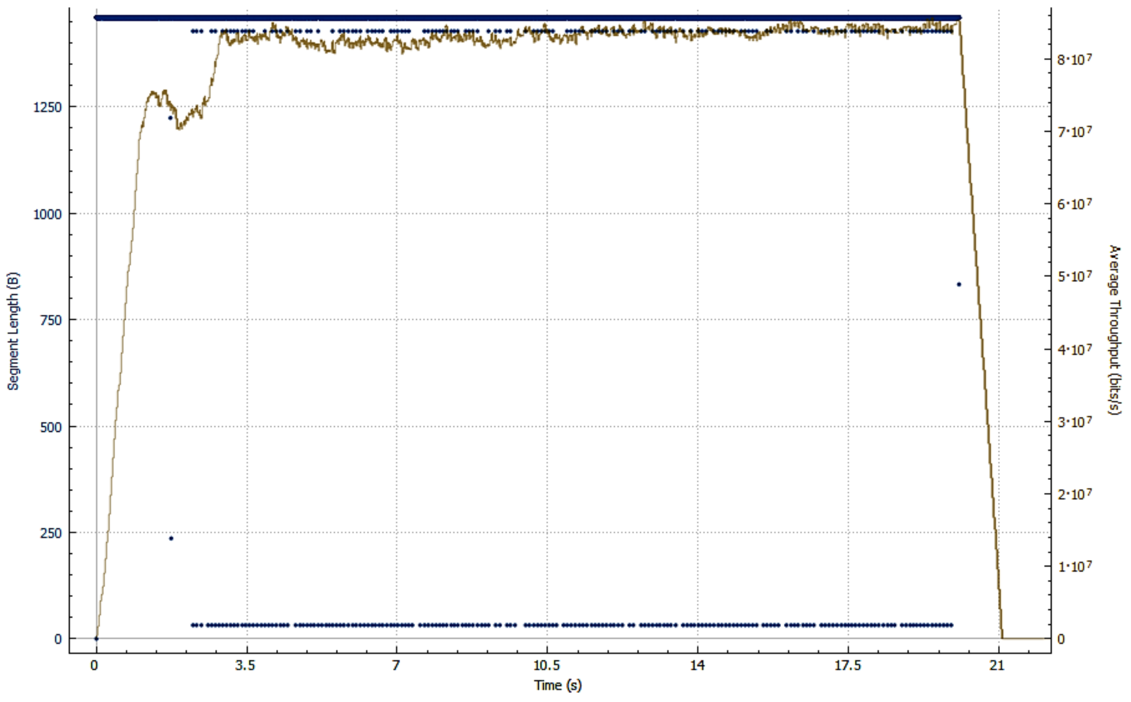


Figure 4.3: Wireshark throughput capture of the Linux node transmitting a 202834 kB file to the Sink. The average throughput level is stable and at a high rate of around 80 Mbps.

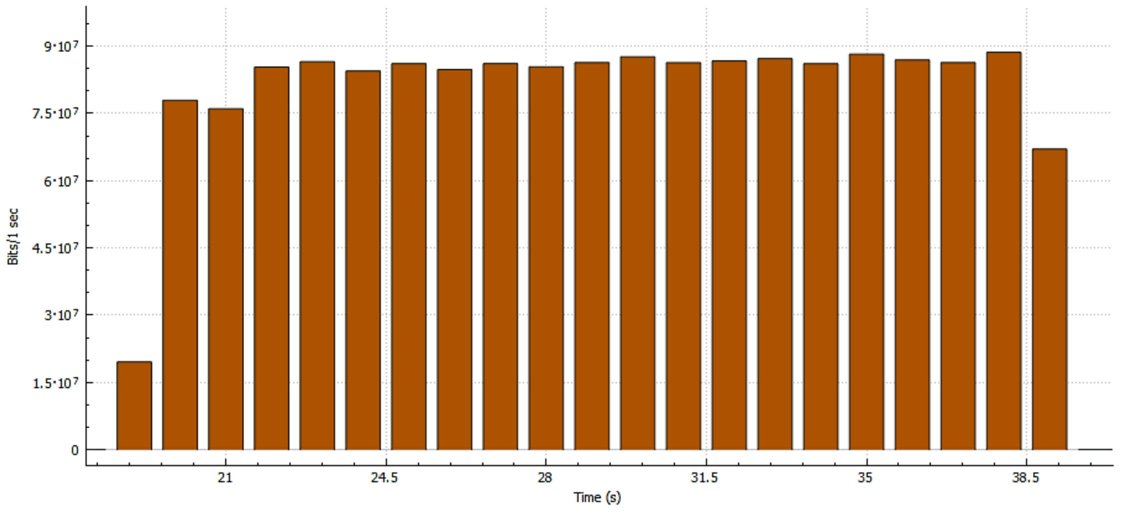


Figure 4.4: Wireshark IO graph of the Linux node transmitting a 202834 kB file to the Sink. It highlights the data rate over the approximate 20 seconds of transmission, where the average rate is around 80 Mbps.

4.3.2 AUTOSAR

To evaluate the performance of the AUTOSAR system, three independent test were performed. The AUTOSAR system's transmitter and receiver part were tested on their own as well as the node forwarding data to the PC it received data from. The transmission time is varying between the measurements due to the fact that when testing the SW parts on their own, a large amount of data was transmitted to allow for observation of the behaviour compared to the forwarding measurement where an image of 10228 kB was used. The Figures 4.5, 4.7, 4.9 and 4.10 display the respective throughput results.

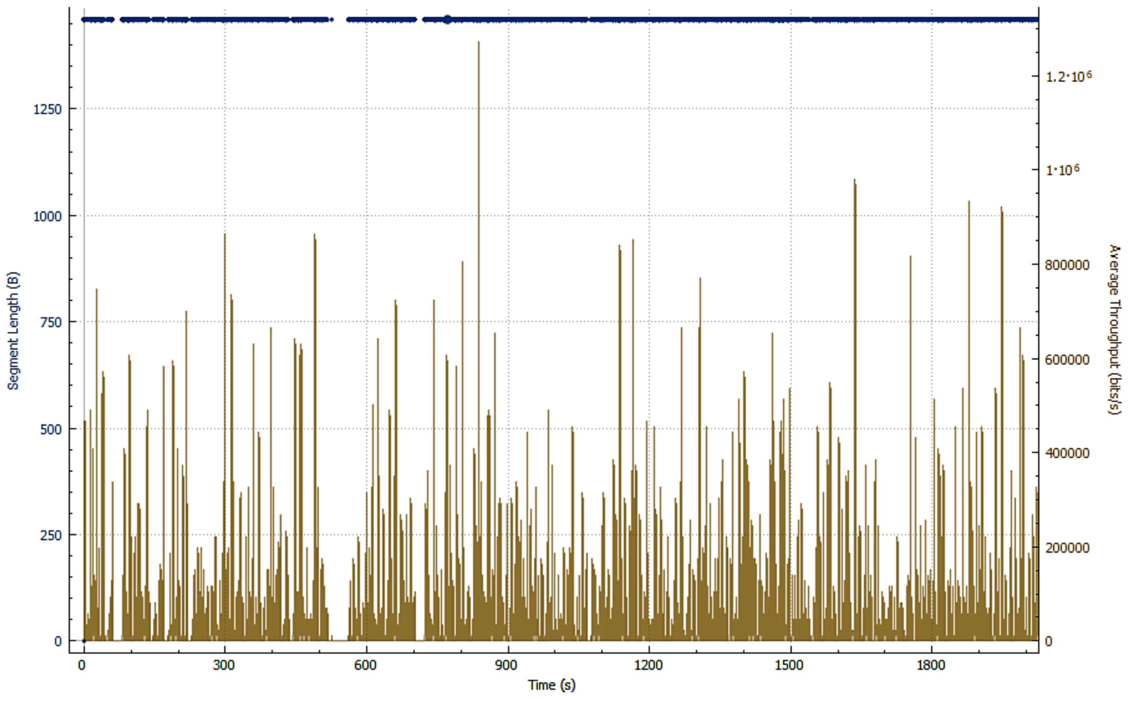


Figure 4.5: Wireshark throughput capture of the AUTOSAR node only transmitting a continuous data flow to the Linux node. The performance is poor, with a low average rate and large fluctuations in the flow of data. Also worth noting is how the transmission rate is zero for several large time periods, the most prominent at around 550 elapsed seconds.

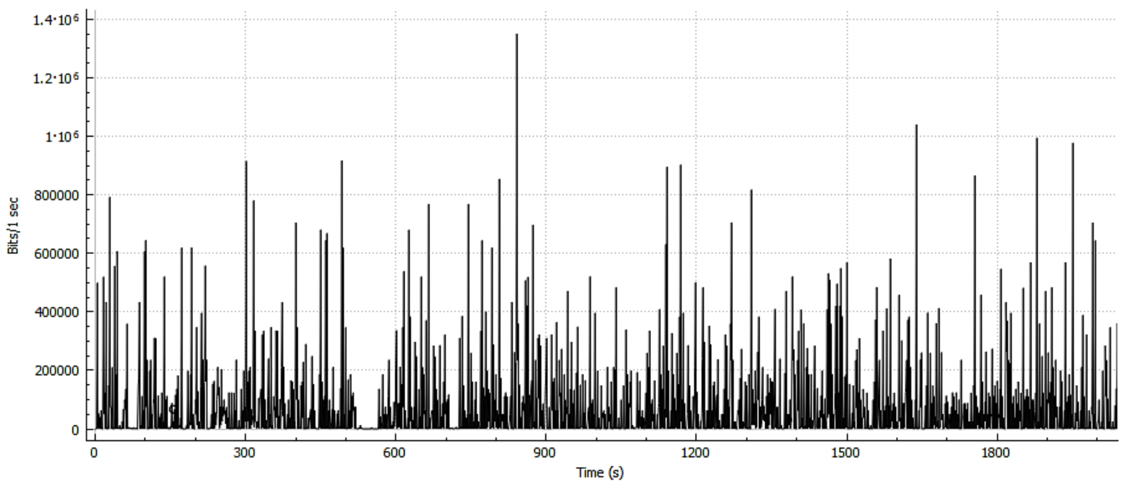


Figure 4.6: Wireshark IO graph of the AUTOSAR node only transmitting a continuous data flow to the Linux node. The data rate is around 0.08 Mbps and there are time periods where it is zero, which are connected to retransmission waiting periods. There are higher peaks of the data rate, which indicates that the system has a higher capacity.

The transmitter part, seen in Figure 4.5 and Figure 4.6, showcases big issues with the AUTOSAR node, namely a highly fluctuating throughput and a low data rate as well as several large time gaps with no transmission. The link did not achieve the maximum capacity of the system, as there are peaks that indicate that higher rates are possible. Debugging of the written code showed that during the time gaps with no transmission, the code is stuck in a segment where the lwIP TCP send buffer is full. The resulting total amount of retransmission packets were around 7%. Possible reasons for this behaviour as well as comments on the low data rate will be discussed in Section 5.1.1.

The measurement on the receiver part showed a slightly more promising result, displayed in Figures 4.7 and 4.8. The throughput behaviour is fluctuating similar to the Tx side, indicating that the capacity has not been reached. There are no time periods with zero transmission, however there are drops down to zero. Upon further inspection, all the negative trends corresponded to TCP retransmissions which will be discussed in Section 5.1.1. The receiving socket window was filled several times during this measurement, each of them resulting in around 1-3 ms delays. The TCP ZeroWindow segments, which are sent by a receiver when its window is full, did not stack up so that there were many of them in a row. However, throughout the transmission, the window was always at or below 50% of its capacity, which indicates that the application fetching data from the receiving window operated slower than what would have been optimal. The main delay is however connected to the many retransmissions. Besides these retransmissions, there also occur many so-called fast retransmissions which are triggered by at least 3 duplicate acks. Of the whole measurement, the total number of retransmission packets were around 9.5% and out of those only around 28% were fast retransmissions. If these would have been more common, the performance should have been increased significantly as they correlate to a much smaller transmission delay.

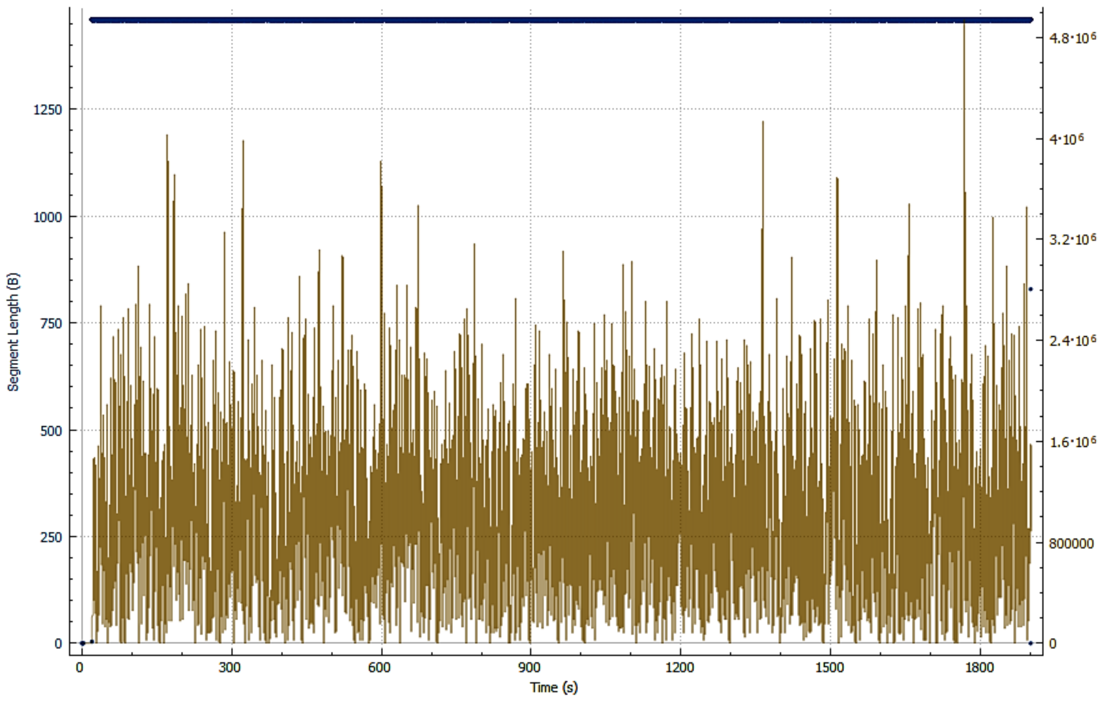


Figure 4.7: Wireshark throughput capture of the AUTOSAR node only receiving a continuous flow of data. The performance is mediocre, with an average receiving rate around 1 Mbps and large fluctuations in the flow of data. Worth noting are the drops down to zero rate, which correlated to when TCP retransmissions occurred.

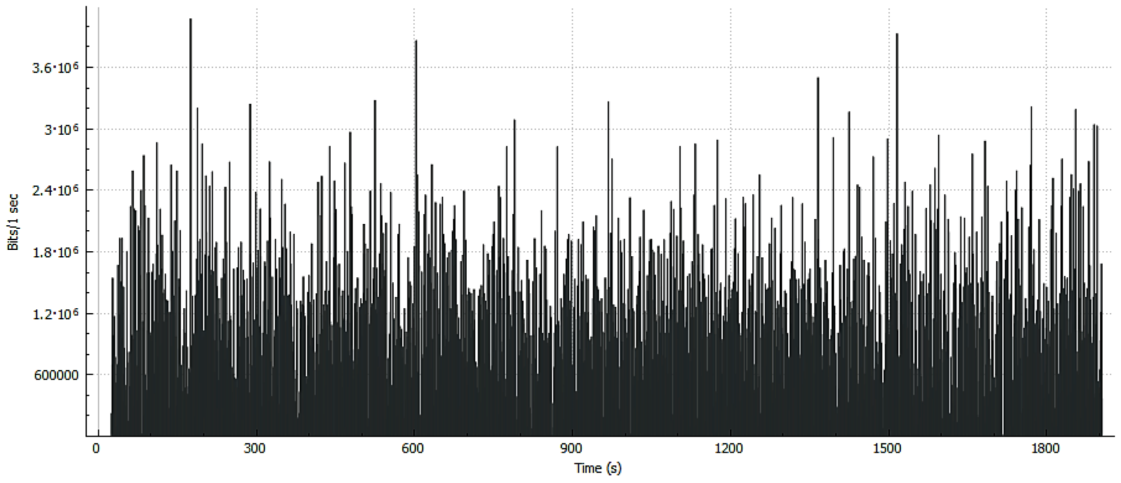


Figure 4.8: Wireshark IO graph of the AUTOSAR node only receiving a continuous data flow. The average rate is around 1 Mbps, however an interesting note is that there are higher peaks indicating that it has not reached its max capacity.

Before the full link was tested, a measurement on the forwarding application SW was performed. Only the throughput feature was used, see Figures 4.9 and 4.10, as the packet lengths were the interesting result. The figures represent the receiving and forwarding throughput respectively. In both figures, the negative trends in data rate again correspond to the frequently occurring retransmissions. With the forwarding SW running, it also resulted in the relatively small window of the AUTOSAR node being filled faster than it is emptied by the forwarding segment. This resulted in the receiving window being completely filled multiple times. This, in turn, results in the receiving side experiencing so-called ZeroWindowProbe packets, with the purpose of the sender checking whether the receiver has freed some space in its window. These are the packets with zero length visible in Figure 4.9. The other packets, in the same figure, that has a lesser length than 1460 are so-called TCP Window Full packets, which as the name suggests are sent with a length corresponding to the remaining space in the receiver window and thereby capping the window.

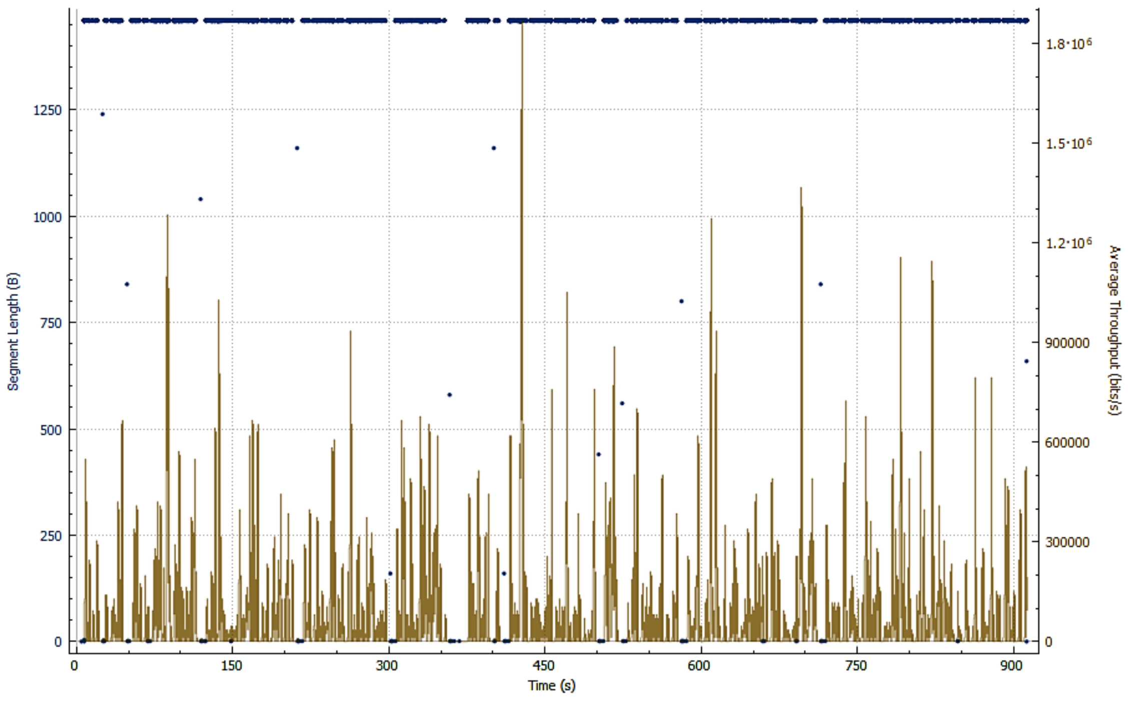


Figure 4.9: Wireshark throughput capture of the AUTOSAR node when receiving a data flow that is further forwarded. The links performance is bad, with a low average rate and long periods with no transmissions. The packet lengths are not stable at the maximum payload size of 1460 because of the receiving window getting filled. Then packets are sent with sizes that match the space left in the window. The packets sent with zero length are so-called ZeroWindowProbes, with the purpose of checking whether the receiver has new space in its window.

The reason why the packet lengths heavily vary in Figure 4.10, is due to the fact that the application SW continuously fills one out of two receiving buffers that the forwarding segment operates on. There is no guarantee that when the forwarding segment gets access to the buffers, that they contain a size which is a multiple of 1460. It will therefore transmit as many packets of size 1460 as possible, as well as the remaining bytes of the buffer before it receives a new buffer to operate on. The Nagle algorithm did therefore not affect the last bytes in each operating buffer, which is not an optimal usage of the bandwidth.

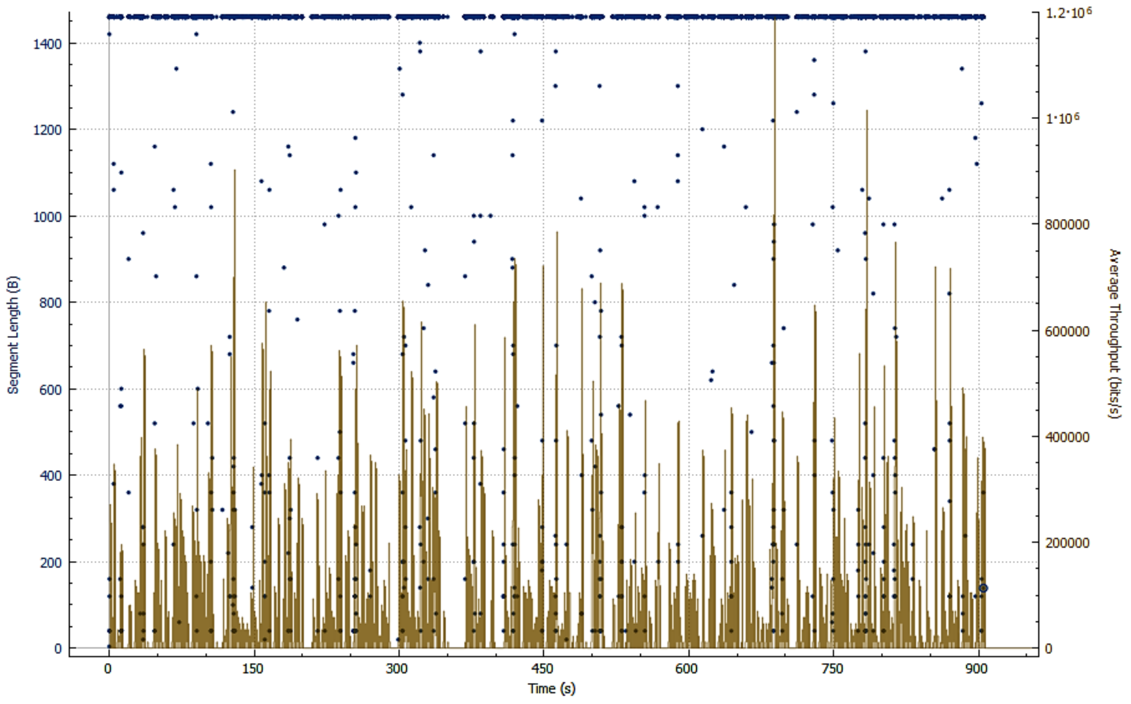


Figure 4.10: Wireshark throughput capture of the AUTOSAR node when forwarding a received data flow. The links performance is bad, as the rate is low as well as long periods with no transmissions. The packet lengths are not stable at the maximum payload size of 1460 because of the application SW not always buffering sizes that are multiples of 1460.

4.3.3 Full link

Finally, measurements on the full link (i.e. PC → AUTOSAR → Linux → PC) were performed. The result is displayed in Figure 4.11. The good performance of the Linux node did not outweigh the bad performance of the AUTOSAR node, as the performance of the full link is poor. The negative tendencies that the AUTOSAR node exhibited is present, a very fluctuating throughput with an average rate of around 110 kbps and long periods with no transmission. When these periods are very long, it results in the initial send program crashing. During large transmissions this occurred frequently, making the link very unreliable.

The negative trends are corresponding to retransmissions and the forwarding segment having a full lwIP TCP send buffer. An interesting behaviour is that of the packet lengths, which occur very frequently with some small lengths. As the forwarding segment will transmit the remaining bytes left in the buffer, the scattered result of different packet lengths is expected. However, the frequent pattern of small lengths that can be observed is due to something else. It looks similar to the behaviour observed in 4.3 and upon further inspection, each of these packets also contained a PSH flag. It however differs somewhat, since it displays multiple levels of the PSH packets and only levels at small lengths. The latter can again be explained by the forwarding segment's behaviour, which differs to the Linux scenario where packets were filled in a buffer of a size multiple of 1460.

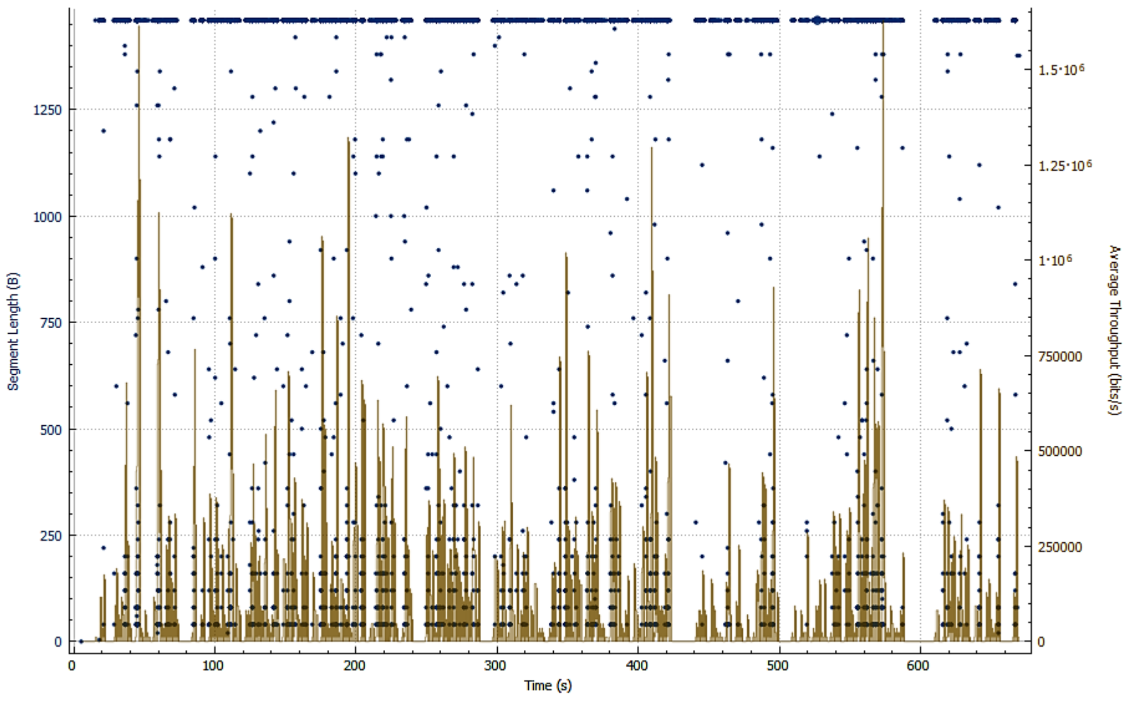


Figure 4.11: Wireshark throughput capture of the AUTOSAR node forwarding its received data to the Linux node. Poor performance is displayed as the throughput is highly fluctuating, with a low average rate of around 110 kbps and with long periods of no transmission.

4.4 OS load

A measurement on the idle time of the AUTOSAR OS was performed at runtime to determine how big of a load it was experiencing. It was measured by extracting two time values, one describing the total time of the measurement and one the idle time. The load was calculated by relating these two to each other, i.e. how much of the total time consisted of idle time.

The load was measured on the AUTOSAR QRx since it was the source of poor performance in the link. It was measured during five different operations. One where the node only looped the receiver part of the software, not doing any processing or forwarding. Likewise, one operation looped only the transmitter part. Two of the modes involved both parts of the SW running, one case when they were running independent of each other and one where they were connected so that the Tx forwards the Rx data. The last mode was with no SW part active and only the underlying OS tasks running. Table 4.2 displays the results.

Table 4.2: OS Load Measurements.

Operation	**Load**	**Uncertainty span**
AUTOSAR Rx only	50 %	±10 %
AUTOSAR Rx & Tx	65 %	±10 %
AUTOSAR Forwarding	75 %	±10 %
AUTOSAR Tx only	24 %	±10 %
Background tasks	10 %	±5 %

The measurements indicate that the OS is experiencing more workload when both of the SW parts are active. It is interesting that when forwarding, the code execution is halted even further compared to when independently transmitting and receiving. This means that besides any buffering issue prominent in the stack, there could also be a timing/delay issue present when swapping between the Tx and Rx tasks.

5 Discussion

Interpretations of the obtained results, as well as an evaluation of the implementation decisions and potential future improvements, will be discussed in this chapter.

5.1 Further analysis of the results

Further analysis of the measurements of the communication link will be discussed. As the reasons behind some of the errors are unclear, the reasonings are of a hypothetical nature.

5.1.1 AUTOSAR

The measurement results of the communication link highlighted several problems with the AUTOSAR implementation. They were mainly located in the implementation of the TCP/IP module, which employed a lwIP solution. As a result of this, two code implementations had to be made to the module i.e. the segment referred to as the TCP throttling and transmission check. These additions seemed to alleviate the original problems, but could likewise be the cause for new ones not encountered yet. Without the throttling segment, the TCP receiving buffer would overwrite its content with the subsequently received packets and thereby causing severe packet loss on the application level. The implementation made to correct for this should have been present in the TCP/IP module from the start, as it is vital and a core principle of TCP. The manual implementation could potentially add delay as it was corrected for at a relative high level of the source code.

Furthermore, the segment regarding the TCP transmission check did not verify why a transmission error had occurred. It only added a solution that would ensure that the callback of the transmission function would match the actual event. If an error occurred, it would now show in the return value and allow for another transmission attempt. However, since these transmission errors occurred during substantial periods, the implemented solution halted the code execution as it was stuck in a transmission loop. Compared to the throttling segment, this implemented solution is more questionable. However, for the purpose of getting actual complete measurements of the system, it was required. If the transmission errors no longer occur, the transmission check segment should be removed as it will then simply add a small delay.

The reason behind the errors leading to all the retransmissions is still unclear, and the following reasonings are therefore only hypothetical. There could be the same underlying reason to both the TCP sender buffer filling up and not transmitting as the regularly occurring transmission errors. However, it is more likely that there are multiple underlying issues with the platform implementation. It is possible that the performed measurements put a strain on the platform implementation not tested before, thus exposing some flaws or bugs in the implementation. As there are many implementation choices that can be made when implementing an AUTOSAR platform, there is a possibility that some of these choices affected the TCP module in unexpected ways. Another theory could be that the entire AUTOSAR software implementation is too slow, resulting in the hardware buffers filling up before the software fetches the content. This could result in either that packet being overwritten or simply dropped when the next packet arrives.

Later versions of AUTOSAR and the ArcCore implementation are available, however, no obvious correction of the mentioned issues could be found in the new configuration files. Although, as there is a big structural change in the newer ArcCore versions, it is possible that these issues have been handled elsewhere. A complex driver of the lwIP, instead of routing it via the TCP/IP module, could possibly have alleviated some of the issues as well, since it would be a more straightforward implementation to the AUTOSAR platform.

Worth to mention, in context with AUTOSAR and Ethernet, is that there exists a restricted amount of previous published work performed on a complete Ethernet link, working with TCP/IP. A majority of the content available is either developer specifications, companies' complete solutions or articles speculating how this may work. None of the mentioned content contributes to experience in working with these systems and the problems which may occur. It is also highly dependent on what AUTOSAR platform implementation used, as platforms may vary structurally.

5.1.2 PSH packets

The measurements displayed a peculiar behaviour in the transmission where multiple packets in a coherent pattern had the PSH flag set. As the purpose of the PSH flag is to force the receiving buffer to release its content to the application layer, there should be no harm in observing packets with this flag. The reason why these packets occur is still unclear, as it was not implemented in the written software. A theory is that it coincides with the TCP delay efficiency concepts. There could potentially be a specification that requires a PSH flag after a specific amount of transmitted bytes or possibly a specification that preemptively tries to prevent the receiving buffer from filling up.

5.1.3 OS load

The load measurements highlighted a difference between transmitting and receiving independent from each other compared to when they were interconnected, where the latter was more exhausting. Why there is a difference between the two cases must be related to the written code. The most likely reason behind this is a slight timing delay. One scenario could be that the Rx and Tx segments were both trying to access the TCP main function at intervals closer to each other than when they were running independently. If the two segments simultaneously want the access, one of the segment will have to wait. The code execution will then become more stacked and directly shift to the other segment, with no idle time in between.

These result also indicate a potential issue, as the TCP main function task could be locked out of execution due to the scheduler being occupied by background tasks. There were potentially many of these background tasks present in the SW, which were unnecessary and could have been removed. Furthermore, AUTOSAR can be configured to change the different OS priorities of specified tasks. Only small changes to this were made and with no apparent result, there is a possibility that this could have been further optimized.

Worth to mention is also that the measurements have a high uncertainty span. This was due to fluctuating measurements, of which the mean was calculated, dependent on manually halting the code execution to retrieve the clock values.

5.2 Improvements & further development

If only observing the measurement results presented in this thesis, it would imply that Ethernet solutions cannot simply be transitioned from other areas. The main issues seem to lie with the AUTOSAR platforms compatibility complications. Prospect for the future indicates that the implementation of Automotive Ethernet would remove the bandwidth limitations that the current bus technologies in vehicles face today. However, as the measurement results indicate, there is much work required in order to piece together an Ethernet backbone network in an automotive scenario. There is a strong possibility that there are other protocols needed than what was used in the thesis, in order to complete an Automotive Ethernet implementation. Thus, a few potential improvements and further development topics will be briefly highlighted.

5.2.1 Adaptive AUTOSAR Platform

The development of the automotive industry, and the autonomous drive section in particular, implies an ever-increasing bandwidth requirement of on-vehicle networks. This is the reason Ethernet was introduced to the automotive industry and even though the classic platform supports Ethernet, it was designed with the legacy communication technologies in mind. It has not been optimized for Ethernet and therefore does not utilize the full potential and capability of Ethernet-based communication. However, the new Adaptive AUTOSAR Platform has better potential to utilize this. Along with better support for multicore processors, the adaptive platform could be the solution to the Automotive Ethernet implementations.

5.2.2 ArcCore version

Many of the problems that occurred with the AUTOSAR implementation could be due to the fact that the ArcCore version used in the thesis is too outdated. As the AUTOSAR standard is continuously evolving, so are the different platform implementations. The latest version is of course preferred, as it better indicates the current situation of the industry, and as mentioned could have solved the issues present in the platform implementation used. A drawback that the AUTOSAR standard implies is that it is a complex architecture to implement. It alleviates the compatibility issues between different SW and HW, but any implementation requires the whole architecture.

5.2.3 BroadR-Reach

As the acquiring of BroadR-Reach technology to use for the practical model was deemed outside the scope of the thesis, the 100Base-TX was chosen as a suitable replacement since both technologies can provide up to 100 Mbps. The BroadR-Reach technology mainly has its advantages in an automotive scenario, where it is more resilient to interference and of less cable weight. Therefore, as the implemented scenario was not in a vehicle, the 100Base-TX should prove to be a sufficient replacement. There should only be small differences in theory, however it would have been preferred to have as many components as possible similar to a real scenario in a vehicle.

5.2.4 SOME/IP

The service-oriented approach that SOME/IP employs has very appealing properties for a more reliable communication within a vehicle, with the ability to subscribe to only the information you need. Furthermore, the concept of serializing frames with a SOME/IP header could potentially solve some compatibility issues as the SOME/IP stack would be implemented with the purpose of being a middleware solution. The subscription latencies that occur when initializing new nodes on the network, with the client and service methodology, could be insignificant if it only occurs in controlled scenarios and not during runtime.

5.2.5 TSN

An issue with Automotive Ethernet is that, even though it will solve some of the bandwidth demands, it is not suitable for time-critical communication needed for autonomous drive applications. Even the addition of SOME/IP would most likely not suffice to provide a time-critical enough communication, as TCP/IP, which consists of nondeterministic properties, still would have to be employed. However, what TSN offers is to skip the higher layer protocols and treat a vehicle like a LAN. By only transmitting Ethernet frames and with the defined scheduling techniques included in the TSN standard, it should produce a time-critical communication that can ensure that the most vital data is successfully prioritized and transmitted in the network. This is of the highest importance for autonomous drive applications, as the safety applications that will depend on the communication technique must be completely foolproof.

As mentioned and displayed by the results, during the project there were issues connected to the TCP/IP module. If the communication would have been based on a TSN implementation instead, the outcome of the measurements might have been indicating a more positive result. Even in a TSN network, the legacy Ethernet communication is included as the BE traffic and could therefore function well as the backbone methodology. Although, further compatibility implementations would probably be needed to include the bus technologies.

Furthermore, a TSN network implies advanced requirements on devices in the network. The CUC and CNC entities have to be present for example. It may be possible to implement a more powerful ECU in the network, holding gateway properties as well as handling the CUC and CNC. Nevertheless, designing the schedule and choosing the appropriate TSAs will be a complex task and perhaps situational to specific applications. As the TSN standard is evolving, it would have been interesting to perform further studies and implementation of the technology.

6
Conclusion

In this thesis, the concept of Automotive Ethernet has been investigated in terms of the construction and testing of a communication link as well as a review of the related technologies in the automotive industry today. From measurements on the communication link, results suggest that the implementation of Automotive Ethernet in coherence with AUTOSAR seems problematic. When the link was subject to larger data loads, it experienced severe performance issues. The communication halted for multiple seconds and the achieved data rate was significantly below the capacity. With the measured performance, Ethernet implementations displaying these results would serve no purpose as the already existing bus technologies perform at a higher rate and are more reliable. However, the results could heavily depend on the platform implementation used in this thesis. The overall concept of Ethernet as a backbone network, in coherence with high-performing bus technologies, seems promising.

Furthermore, with the addition of new standards such as SOME/IP and TSN, systems may experience significant improvements. The fact that TSN operates only at the lower OSI-layers and that an automotive network is comparable to a LAN, the method should allow for less application delay etc. by excluding standard protocols such as TCP/IP. Compared to TCP/IP, TSN can also guarantee that prioritized data is transmitted without delay and in a deterministic way. Two properties that are crucial in time-critical systems. Therefore, with a continued development of the TSN standard, it seems like a promising extension to the field of autonomous drive and it bodes well for the future.

There is also the Adaptive AUTOSAR platform, which is a new standard currently being developed that exhibit promising aspects. A system running on this OS, better adapted to an Ethernet implementation and perhaps in unison with protocols such as TSN and SOME/IP, may be the much-needed breakthrough for the future of autonomous drive. For further research within this field, we recommend looking into the possibility of implementing lwIP as a standalone module. Furthermore, exploring the newer Adaptive platform in coherence with the TSN standard.

Bibliography

[1] Stefan Bunzel. “AUTOSAR – the Standardized Software Architecture”. In: *Informatik-Spektrum* 34.1 (2010), pp. 79–83. DOI: `10.1007/s00287-010-0506-7`.

[2] AUTOSAR cooperation. *AUTOSAR*. 2018. URL: `https://www.autosar.org`. Accessed on: 2018-01-22.

[3] Dr. Lars Völker. *Scalable service-Oriented MiddlewarE over IP (SOME/IP)*. 2018. URL: `http://some-ip.com/`. Accessed on: 2018-01-22.

[4] VECTOR cooperation. *Introduction to Automotive Ethernet*. 2018. URL: `https://elearning.vector.com/index.php?&wbt_ls_seite_id=1603254&root=835866&seite=vl_automotive_ethernet_introduction_ko`. Accessed on: 2018-01-22.

[5] M. Szarvas, U. Sakai, and J. Ogata. “Real-time Pedestrian Detection Using LIDAR and Convolutional Neural Networks”. English. In: IEEE, Jan. 2006, pp. 213–218. ISBN: 490112286X;9784901122863;

[6] VECTOR cooperation. *Introduction to CAN*. 2018. URL: `https://elearning.vector.com/vl_can_introduction_en.html`. Accessed on: 2018-01-22.

[7] Li Ran et al. “Design method of CAN BUS network communication structure for electric vehicle”. English. In: 2010, pp. 326–329. ISBN: 9781424490387;1424490383;

[8] Robert.S, Dr. Jayasudha J.S, and Anurag. “TCP/IP Stack Implementation for Communication over IP with AUTOSAR Ethernet Specification”. In: *2013 International Journal of Engineering and Innovative Technology (IJEIT)*. Vol. 3. 1. July 2013, pp. 176–179.

[9] Mikael Degermark et al. “Small Forwarding Tables for Fast Routing Lookups”. In: *SIGCOMM Comput. Commun. Rev.* 27.4 (Oct. 1997), pp. 3–14. ISSN: 0146-4833. DOI: `10.1145/263109.263133`. URL: `http://doi.acm.org/10.1145/263109.263133`.

[10] P. Koopman and T. Chakravarty. “Cyclic redundancy code (CRC) polynomial selection for embedded networks”. English. In: Palazzo dei Congressi, Florence, Italy: IEEE, 2004, pp. 145–154. ISBN: 0769520529;9780769520520;

[11] Greg Minshall et al. “Application Performance Pitfalls and TCP’s Nagle Algorithm”. In: *SIGMETRICS Perform. Eval. Rev.* 27.4 (Mar. 2000), pp. 36–44. ISSN: 0163-5999. DOI: `10.1145/346000.346012`. URL: `http://doi.acm.org.proxy.lib.chalmers.se/10.1145/346000.346012`.

[12] VECTOR cooperation. *Introduction to Serial Bus Systems in Motor Vehicles*. 2018. URL: `https://elearning.vector.com/index.php?wbt_ls_kapitel_id=507950&root=378422&seite=vl_sbs_introduction_en`. Accessed on: 2018-02-07.

[13] Kirsten Matheus and Thomas Königseder. *Automotive Ethernet*. English. Cambridge: Cambridge University Press, 2015. ISBN: 9781107057289; 1107057280; 1107183227; 9781107183223.

[14] Nan Liang and Dobrivoje Popovic. "2 - The {CAN} bus". In: *Intelligent Vehicle Technologies*. Ed. by Ljubo Vlacic, Michel Parent, and Fumio Harashima. Automotive Engineering Series. Oxford: Butterworth-Heinemann, 2001, pp. 21–64. ISBN: 978-0-7506-5093-9. DOI: `https://doi.org/10.1016/B978-075065093-9/50004-9`. URL: `https://www.sciencedirect.com/science/article/pii/B9780750650939500049`.

[15] Qiangsheng Ye. "Research and application of CAN and LIN bus in automobile Network System". In: *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*. Vol. 6. Aug. 2010, pp. V6-150-V6-154. DOI: `10.1109/ICACTE.2010.5579409`.

[16] National Instruments Corporation. *FlexRay Automotive Communication Bus Overview*. 2016. URL: `http://www.ni.com/white-paper/3352/en/`. Accessed on: 2018-04-25.

[17] National Instruments Corporation. *Introduction to the Local Interconnect Network (LIN) Bus*. 2016. URL: `http://www.ni.com/white-paper/9733/en/`. Accessed on: 2018-04-25.

[18] C. Gabriel and H. Horia. "Integrating sensor devices in a LIN bus network". English. In: vol. 2003-. IEEE, 2003, pp. 150–153. ISBN: 2161-2528.

[19] MOST cooperation. *Media Oriented Systems Transport*. 2018. URL: `https://www.mostcooperation.com/`. Accessed on: 2018-02-07.

[20] Andrzej Sumorek and Marcin Buczaj. "The evolution of "Media Oriented Systems Transport" protocol". In: *TEKA Commission of Motorization and Energetics in Agriculture* 14 (Aug. 2014), pp. 115–120.

[21] Otto Strobel, Ridha Rejeb, and Jan Lubkoll. "Communication in automotive systems: Principles, limits and new trends for vehicles, airplanes and vessels". In: *2010 12th International Conference on Transparent Optical Networks* (June 2010). ISSN: 2162-7339. DOI: `10.1109/icton.2010.5549163`.

[22] Vector Informatik. *Media Oriented Systems Transport (MOST)*. 2018. URL: `https://vector.com/vi_most_en.html`. Accessed on: 2018-04-26.

[23] Andreas Grzemba. *MOST: The Automotive Multimedia Network; from Most25 to Most150*. Franzis, 2011. ISBN: 978-3-645-65061-8.

[24] Y. S. Lee, J. H. Kim, and J. W. Jeon. "FlexRay and Ethernet AVB Synchronization for High QoS Automotive Gateway". In: *IEEE Transactions on Vehicular Technology* 66.7 (July 2017), pp. 5737–5751. ISSN: 0018-9545. DOI: `10.1109/TVT.2016.2636867`.

[25] Broadcom Corporation. *BroadR-Reach Physical Layer Transceiver Specification For Automotive Applications*. 2014. URL: `http://www.ieee802.org/3/1TPCESG/public/BroadR_Reach_Automotive_Spec_V3.0.pdf`. Accessed on: 2018-04-25.

[26] Shanker Shreejith et al. "VEGa: A High Performance Vehicular Ethernet Gateway on Hybrid FPGA". In: *IEEE Transactions on Computers* 66.10 (2017), pp. 1790–1803. DOI: `10.1109/tc.2017.2700277`.

[27] Till Steinbach, Franz Korf, and Thomas Schmidt. “Real-time Ethernet for Automotive Applications: A Solution for Future In-Car Networks”. In: *IEEE International Conference on Consumer Electronics* (Sept. 2011), pp. 216–220. DOI: 10.1109/ICCE-Berlin.2011.6031843.

[28] Lucia Lo Bello. “The Case for Ethernet in Automotive Communications”. In: *SIGBED Rev.* 8.4 (Dec. 2011), pp. 7–15. ISSN: 1551-3688. DOI: 10.1145/2095256.2095257.

[29] IEEE. *Time-Sensitive Networking (TSN) Task Group.* 2018. URL: https://1.ieee802.org/tsn/. Accessed on: 2018-02-14.

[30] Y. S. Lee, J. H. Kim, and J. W. Jeon. “FlexRay and Ethernet AVB Synchronization for High QoS Automotive Gateway”. In: *IEEE Transactions on Vehicular Technology* 66.7 (July 2017), pp. 5737–5751. ISSN: 0018-9545. DOI: 10.1109/TVT.2016.2636867.

[31] Cisco Systems Inc. *Time-Sensitive Networking: A Technical Introduction.* 2017. URL: https://www.cisco.com/c/dam/en/us/solutions/collateral/industry-solutions/white-paper-c11-738950.pdf. Accessed on: 2018-05-20.

[32] Avnu Alliance. *Avnu Alliance.* 2018. URL: http://avnu.org/. Accessed on: 2018-05-22.

[33] T. Steinbach et al. “Tomorrow’s In-Car Interconnect? A Competitive Evaluation of IEEE 802.1 AVB and Time-Triggered Ethernet (AS6802)”. In: *2012 IEEE Vehicular Technology Conference (VTC Fall).* Sept. 2012, pp. 1–5. DOI: 10.1109/VTCFall.2012.6398932.

[34] Sune Mølgaard Laursen, Paul Pop, and Wilfried Steiner. “Routing Optimization of AVB Streams in TSN Networks”. In: *SIGBED Rev.* 13.4 (Nov. 2016), pp. 43–48. ISSN: 1551-3688. DOI: 10.1145/3015037.3015044. URL: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/3015037.3015044.

[35] W. Steiner et al. “Next generation real-time networks based on IT technologies”. In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA).* Sept. 2016, pp. 1–8. DOI: 10.1109/ETFA.2016.7733580.

[36] Vector Informatik. *Basic Software and RTE.* 2016. URL: https://elearning.vector.com/index.php?wbt_ls_kapitel_id=1045000&root=378422&seite=vl_autosar_introduction_en. Accessed on: 2018-04-27.

[37] Arccore AB. *Introduction to Complex Drivers.* 2014. URL: http://dev.arccore.com/public/user-doc/UD441x/Introduction-to-Complex-Drivers_28607110.html. Accessed on: 2018-04-26.

[38] Nico Naumann. “AUTOSAR Runtime Environment and Virtual Function Bus”. MA thesis. Postdam, Germany: Department for System Analysis and Modeling, Hasso-Plattner Institute for IT-Systems Engineering.

[39] AUTOSAR. *SOME/IP Protocol Specification.* 2016. URL: https://www.autosar.org/fileadmin/user_upload/standards/foundation/1-0/AUTOSAR_PRS_SOMEIPProtocol.pdf. Accessed on: 2018-05-25.

[40] Jan Seyler, Nicolas Navet, and Loïc Fejoz. “Insights on the Configuration and Performances of SOME/IP Service Discovery”. In: *SAE International Journal of Passenger Cars - Electronic and Electrical Systems* 8.1 (Apr. 2015), pp. 124–

129. ISSN: 1946-4622. DOI: `https://doi.org/10.4271/2015-01-0197`. URL: `https://doi.org/10.4271/2015-01-0197`.

[41] AUTOSAR. *Explanations of Adaptive Platform Design*. 2017. URL: `https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_EXP_PlatformDesign.pdf`. Accessed on: 2018-04-25.

[42] R Pallierer and B Schmelz. *Combine AUTOSAR Standards for High-Performance In-Car Computers*. 2017. URL: `http://innovation-destination.com/2017/12/13/combine-autosar-standards-high-performance-car-computers/`. Accessed on: 2018-04-25.

[43] ArcCore cooperation. *ArcCore*. 2018. URL: `https://www.arccore.com/`. Accessed on: 2018-02-08.