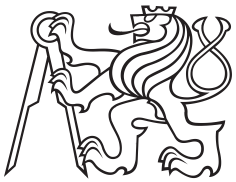
Master's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Measurement

SOME/IP Implementation for Testing

Bc. Jakub Hortenský

Supervisor: Ing. Jan Sobotka, Ph.D.
May 2023

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Hortenský Jakub** Personal ID number: **483471**

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

Department / Institute: **Department of Measurement**

Study program: **Cybernetics and Robotics**

II. Master’s thesis details

Master’s thesis title in English:

SOME/IP Implementation for Testing

Master’s thesis title in Czech:

Implementace SOME/IP protokolu pro testování

Guidelines:

1. Familiarize yourself with the SOME/IP protocol (part of AUTOSAR 4.3).
2. Do research for a method of formal description of offered services.
3. Implement import of selected description.
4. Parametrize general client based on the imported description for specific SOME/IP services.
5. The result will be software acting as a client, a server is optional.
6. Don’t forget to document the code.
7. Demonstrate your results by suitable SOME/IP demo communication.

Bibliography / sources:

[1] Matheus, Kirsten, and Thomas Ko nigseder. Automotive Ethernet. Third edition. Cambridge University Press, 2021.
[2] Nicolas Navet, F. and Simonot-Lion, F.: Automotive Embedded Systems Handbook, CRC PressINC, 2009.

Name and workplace of master’s thesis supervisor:

Ing. Jan Sobotka, Ph.D. Department of Measurement FEE

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

Date of master’s thesis assignment: **16.02.2023** Deadline for master's thesis submission: _____

Assignment valid until:
by the end of summer semester 2023/2024

Ing. Jan Sobotka, Ph.D.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

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

.	
Date of assignment receipt	Student’s signature

Acknowledgements

I want to express my profound gratitude to my supervisor, Ing. Jan Sobotka, Ph.D., for his valuable insight, help, and feedback. I would also like to thank doc. Ing. Jiří Novák, Ph.D., for his help with the project. Additionally, I would like to thank my parents and friends for their endless support throughout my entire studies.

Declaration

I hereby declare, that I have worked on my master's thesis separately and that I have listed all the used literature.

In Prague, 26. May 2023

Abstract

The main goal of this master's thesis is the implementation of a client capable of communication using SOME/IP according to AUTOSAR specifications and parametrizing itself based on the imported service descriptions. An auxiliary virtual test server was implemented to validate the client's functionality. The validation was conducted on the virtual test server and, consequently, on the real hardware. The result of this master's thesis complies with the assignment and can be used in further SOME/IP testing.

Keywords: communication, automotive, Ethernet, SOME/IP

Supervisor: Ing. Jan Sobotka, Ph.D.

Abstrakt

Tato diplomová práce se věnuje implementaci klienta schopného komunikovat pomocí protokolu SOME/IP podle AUTOSAR specifikací. Klient je schopen se sám parametrizovat na základě načtených popisů služeb. Pro validaci správného fungování klienta byl implementován pomocný virtuální testovací server. Fuknčnost byla validována na tomto server a následně na reálném hardwaru. Výsledek této diplomové práce souhlasí s zadáním a lze být využit pro další tesování protokolu SOME/IP.

Klíčová slova: komunikace, automobilový průmysl, Ethernet, SOME/IP

Překlad názvu: Implementace SOME/IP Protokolu pro Testování

Contents

Figures

Tables

Chapter 1

Introduction

The automotive industry is responsible for remarkable advancements in transportation in the last century, shaping societies and dreams worldwide. But nowadays, vehicles are not only roaring engines and beautiful curves; they have evolved into complex and sophisticated computers on wheels connected through elaborate networks. These automotive networks provide the means for seamless communication, enabling a wide selection of functionalities for the driver, ranging from safety features to advanced driver-assistance systems (ADAS). There are many network protocols used in modern vehicles (CAN, LIN, FlexRay, etc.); this thesis aims to explore and utilize a new addition, which is gaining prominence, the Service-Oriented Middleware over IP (SOME/IP). This protocol revolutionizes the way the individual units communicate and interact with one another.

The structure of this thesis is organized as follows: Chapter 2 provides a comprehensive overview of the protocols forming the SOME/IP stack, establishing a solid foundation for the subsequent chapters. Chapter 3 delves into the implementation of the client utilizing the SOME/IP protocol to connect to the automotive network and communicate with its devices. Chapter 4 describes the experiments and tests performed by this client. And finally, chapter 5 summarizes the whole thesis and explores potential areas of improvement.

The thesis endeavors to familiarize the reader with the protocols used in the SOME/IP client and its working mechanisms while simultaneously persuading the person reading this of its benefits and the reasons for its increasing usage in modern vehicles.

Chapter 2

Background

The communication between the client and server can be described using an OSI (Open Systems Interconnection) model [7], where each protocol corresponds to a specific layer. Each protocol is built upon the previous, constructing a chain. The chain is then gradually broken down at the receiving side, layer by layer. Additional information about used protocols is written in the following sections; additionally, their assigned layers can be seen in Figure 2.1.

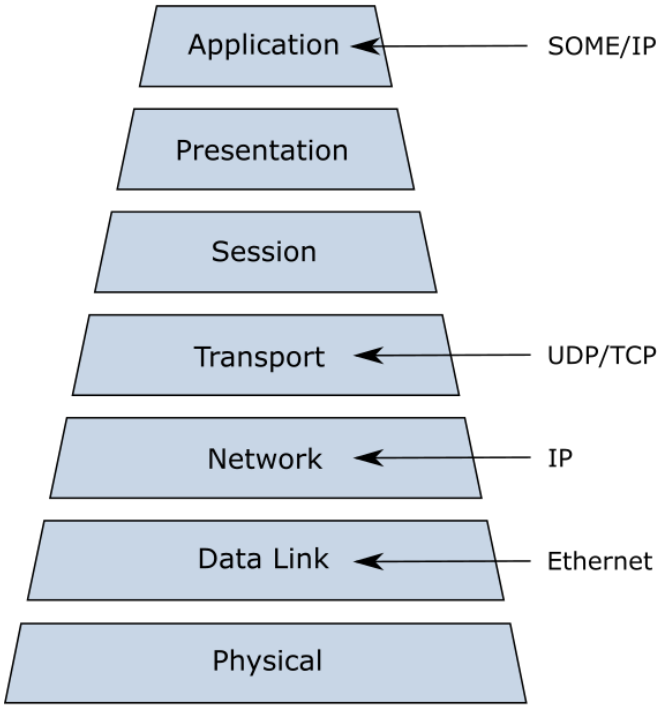


Figure 2.1: OSI model with highlighted protocols

2.1 Ethernet

Ethernet is a widely-used technology enabling devices in the same local area network (LAN) to communicate with each other. Ethernet protocol operates as the first (physical) and the second (link) layer of the OSI model. The link layer is further divided into two individual layers, the Logical Link Control layer and the Medium Access Control layer.

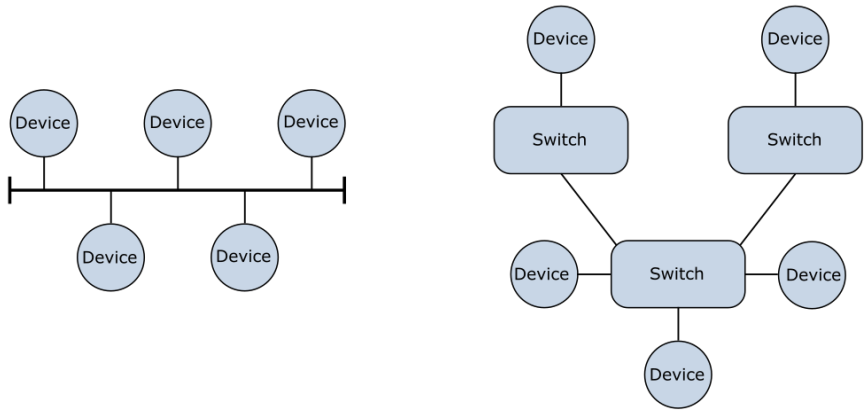
Ethernet can operate on twisted pair copper cables, optical fiber, or coaxial cables, depending on the Ethernet variant. Nowadays, the most common variants used are 100BASE-TX and 1000BASE-T, where both use twisted pair cables with a typical termination using 100 Ω resistors. The name of each specification can be broken down into three segments; the nominal value, the band, and the medium. The nominal value represents speed in megabits per second. The band can be either baseband (BASE), broadband (BROAD), or passband (PASS). In the majority of cases, the baseband is used. The baseband refers to the range of frequencies enveloping the original signal, usually ranging from 0 Hz up to an upper limit. The letters after the dash represent the medium, according to Table 2.1. The optional last letter represents the data encoding method. This last letter can be skipped for unique encoding.

Code	Description	Code	Description
T	twisted pairs	H	plastic optical fiber
T1	single twisted pair	C	twinaxial cable
S	short wavelength (850 nm)	E	extra long wavelength (1500 nm)
L	long wavelength (1300 nm)	F	fiber with various wavelengths

Table 2.1: Ethernet medium codes

The topology of the Ethernet network has changed throughout its existence. The original idea was to use a singular bus to which all the devices would be connected. This topology, while reliable and effective for small networks, is not suited for more extensive networks. In later variants, devices are hidden behind switches that communicate with each other. Switches are devices operating on the second layer of the OSI model, forwarding data only to its destination. However, the connected grid creates loops where frames may run endlessly. This issue is handled by the Spanning Tree Algorithm [3] or by the improved Rapid Spanning Tree algorithm [4]. This algorithm transforms the grid or star topology into the tree network topology. First, a root bridge is selected, which becomes a reference to determine the connection cost between each node and the root. This cost is usually computed using communication speed. After each node has evaluated all its connections, the cheapest route

is preserved, while all the others are terminated (the ports are deactivated for forwarding), as visualized in Figure 2.3.



(a) : Bus topology **(b) :** Star topology

Figure 2.2: Ethernet topologies

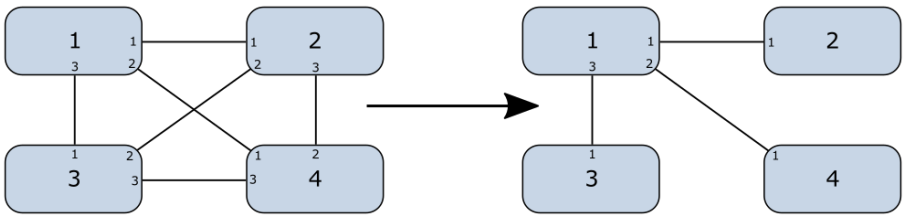


Figure 2.3: Rapid Spanning Tree algorithm visualization

To ensure collision-free communication on a shared medium, Ethernet uses a protocol called Carrier Sense Multiple Access with Collision Detection (CSMA/CD). Before transmitting any data, a node waits and listens to the network for a random amount of time. If the network remains idle for the entire duration, the node begins transmitting while simultaneously listening for any sign of a collision. If a collision is detected, the transmitting stops, and the node returns to the waiting state. The more recent Ethernet variants abandon the use of this protocol in favor of full-duplex switches.

Each Ethernet frame begins with a 7-byte Preamble and a 1-byte Start Delimiter. The bits in each byte of the Preamble switch their value, as showcased in Table 1. This bit alternating allows for bit-level synchronization.

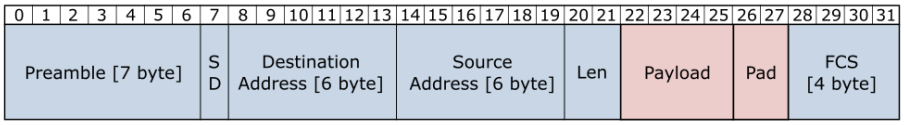


Figure 2.4: Ethernet frame structure

Start Delimiter looks like any other byte of the Preamble apart from the last bit, which is inverted. Following are two 6-byte MAC address fields. The 2-byte length field covers the length of the entire frame. The newer specifications of the ethernet frame change this field to EtherType, which keeps the information about the length, but adds an option to express the type of payload. The padding ensures a minimal frame length of 64 bytes. The last four bytes are occupied by the frame check sequence (FCS). A cyclic redundancy check (CRC) is implemented here with a CCIT-32 generating polynomial.

Field	1-byte value
Preamble	10101010
SD	10101011

Table 2.2: Ethernet preamble bits

Nodes in automotive Ethernet networks are often grouped together into virtual local area networks (VLAN) with shared physical infrastructure. This virtual mask is managed and implemented by the active switches. Frames are tagged according to the VLAN to which they belong. These tags are given to the frame either by the node or the switch they trespass through. Tag is a 4-byte long field fitted between the source address and the length. The information about the frame priority can be a part of the VLAN frame.

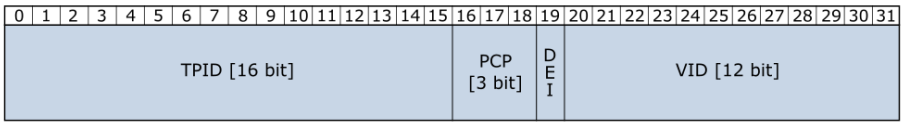


Figure 2.5: VLAN frame

- **TPID** - a 16-bit field indicating that the VLAN tag is present
- **PCP** - a 3-bit field containing the priority of the frame
- **DEI** - a 1-bit field signaling the relevance of the frame
- **VID** - a 12-bit VLAN identifier

The time synchronization is managed by the Precision time protocol (PTP). The PTP master sends a Sync message at time t_1 and a Follow_up message with a record of t_1. The Sync message is received by the PTP slave at t_2. This process is repeated for t_3 and t_4, only this time transaction is initialized by the PTP slave. The transmission delay D and offset O are calculated as shown in Equations 2.1. The delay represents the time it takes the frame to reach the receiver, while the offset is caused by switches and is negative in the reverse direction. It is assumed both of these values to be constants. This process is repeated frequently.

$$\begin{aligned} t_2 &= t_1 + D + O \\ t_4 &= t_3 + D - O \end{aligned} \tag{2.1}$$

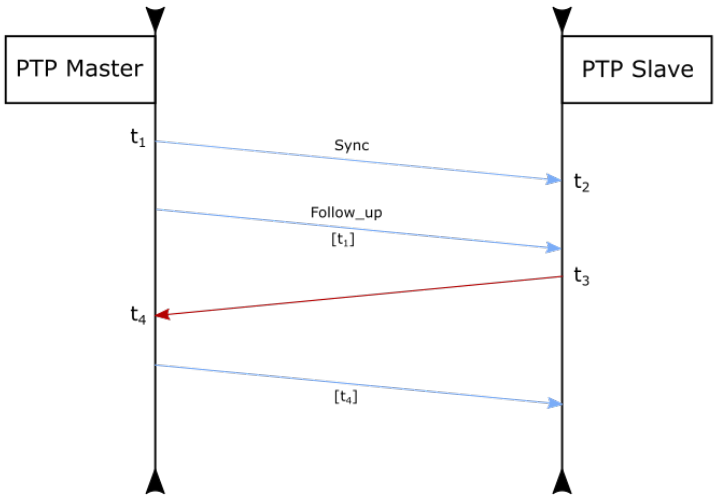


Figure 2.6: PTP protocol

2.1.1 Automotive ethernet

The Ethernet specifications used in the automotive environment are 100BASE-T1 and 1000BASE-T1, enabling the transmitting speeds of 100 Mbit per second and 1000 Mbit per second respectively. To be as lightweight as possible, these versions transmit using only one twisted pair cable while still maintaining full-duplex functionality. This is possible by implementing echo cancellation and decision feedback equalization techniques [13]. The former removes the echo by subtracting the sent signal from the received. The latter is a necessity providing jitter and noise immunity made highly relevant due to the used signal modulation. It uses a 3-level pulse-amplitude modulation (PAM3). The signal is encoded into three levels (-1, 0, 1), each called a

symbol. Bits are grouped together into 3-bit batches, which are mapped onto the pair of symbols defined by PAM3. This process is showcased in Table 2.3. The null pair (0, 0) is used for the beginning and the end of a frame.

3-bit batch	ternary A	ternary B
000	-1	0
001	0	1
010	-1	1
011	0	1
Reserved	0	0
100	1	0
101	0	-1
110	1	-1
111	0	-1

Table 2.3: Ethernet PAM3 symbols

The connection between nodes can be only peer-to-peer with an impedance matching of 100 ohms. The distance between nodes cannot exceed 15 meters, which is a sufficient range for use in cars. One of the nodes claims the role of a master, and the other the role of a slave. This distribution is relevant only for the beginning of the communication.

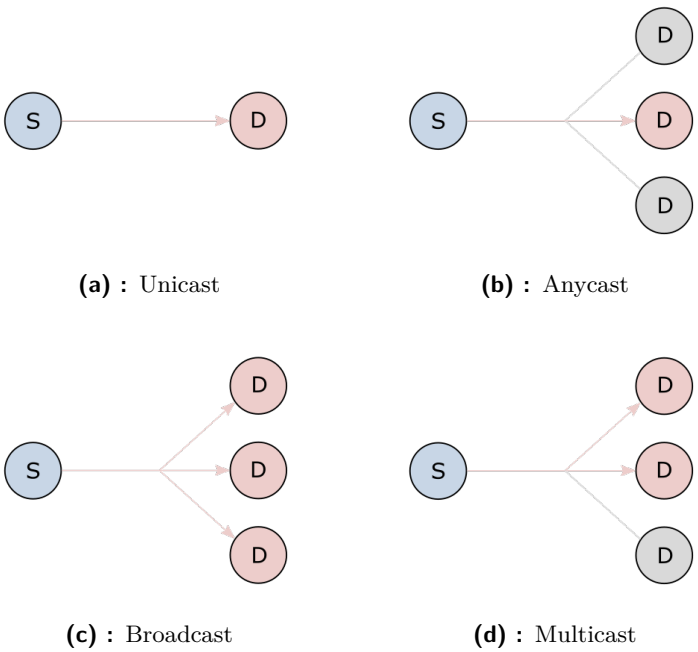
2.2 IP

IP (Internet Procol) was introduced as a network layer in the Internet protocol suite [9]. Its establishment predates the OSI model by a few years, but it can be safely associated with the third OSI layer, the network layer. Its main responsibility is to address and route data packets across the network. It manages this task by inserting an IP header in front of the packet with a record of the source and destination address; therefore, any recipient of this packet knows where the packet came from and where it is heading. This addressing relies on the fact that in the given network, IP addresses are unique for all devices.

IP is a connectionless protocol (message-oriented), e.i., each packet is treated as a single independent unit, and prior to the transmission, no communication channel or stream is created. IP is commonly used in conjunction with transport protocols, such as TCP and UDP, depending on the communication needs.

IP addressing can work in four different modes, depending on the number of devices inside the network and the purpose of the communication. The modes in question are:

- **Unicast** - the connection is peer-to-peer, e.i., only between two devices
- **Anycast** - the message is intended only for one recipient
- **Broadcast** - the message is sent to all recipients inside a network
- **Multicast** - the message is sent to an arbitrary number of recipients



(a) : Unicast **(b) :** Anycast

(c) : Broadcast **(d) :** Multicast

Figure 2.7: IP addressing modes

Currently, two versions of IP are used, IPv4 and IPv6. The primary difference between these two versions is their address space, where IPv6 uses 128-bit addresses instead of 32-bit addresses.

2.2.1 IPv4

IPv4 addresses are 32-bit integer values consisting of four octets separated by periods, e.g., 192.168.1.1. In theory, there are 2^{32} (nearly 4.3 billion) possible addresses; however, many of these are reserved or private. According to IETF (Internet Engineering Task Force) [10], the number of reserved IPv4 addresses is slightly over 324 million. That means the number of available IPv4 addresses is nearly 4 billion. As of this year (2023), the global population is slightly above 8 billion [2]. Even if each person had only one device, there would be a great deficit of addresses. The IPv6 addressing solves this problem.

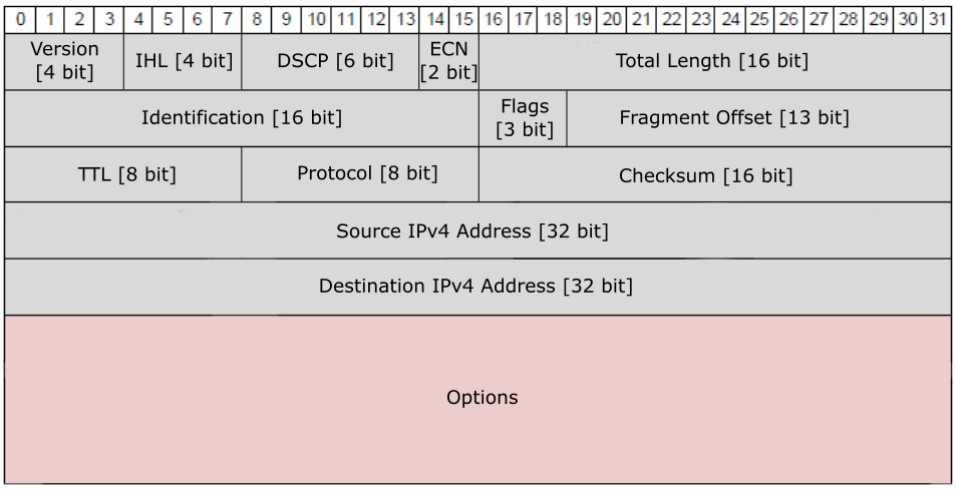


Figure 2.8: IPv4 packet structure

The IPv4 header, visualized in Figure 2.8, consists of 13 mandatory fields and one optional field, Options. The first 32 bits represent the version, the length of the header (IHL), the previous Type of Service field, now DSCP and ECN improving the quality of service, and the total IP packet length. Subsequent 32 bits manage the IP packet fragmentation. The time to live field (TTL) prevents any failures in the event of a routing loop by decrementing this field by one for each arrival at a router. When the value of this field drops to zero, the packet is discarded. The protocol field determines the encompassed protocol carried by the IP packet by assigning each viable protocol a code [1]. The checksum is present in the IP packet for error-checking the header. The subsequent protocols should have error-checking mechanisms of their own. The last 64 mandatory bits belong to the source and the destination IP addresses, respectively.

2.2.2 IPv6

IPv6 is the most recent version of the Internet Protocol, intended to replace IPv4 due to its limited address space. Instead of 32-bit addresses, IPv6 uses 128-bit addresses, capable of accommodating 2^{128} unique addresses (approximately 340 undecillion or $3.4 \cdot 10^{38}$). A small portion of these addresses is already reserved, almost three undecillion (circa 0.88 %), which still leaves a vast quantity of addresses vacant. Another major change is fragmentation, where the IPv6 protocol leaves the fragmentation solely to the application due to security reasons.

The changes in the IPv6 header structure, visualized in Figure 2.9, begin with the omission of the IHL field and the conjunction of DSCP and ECN into one field, the Traffic class. Next is the replacement of the total length

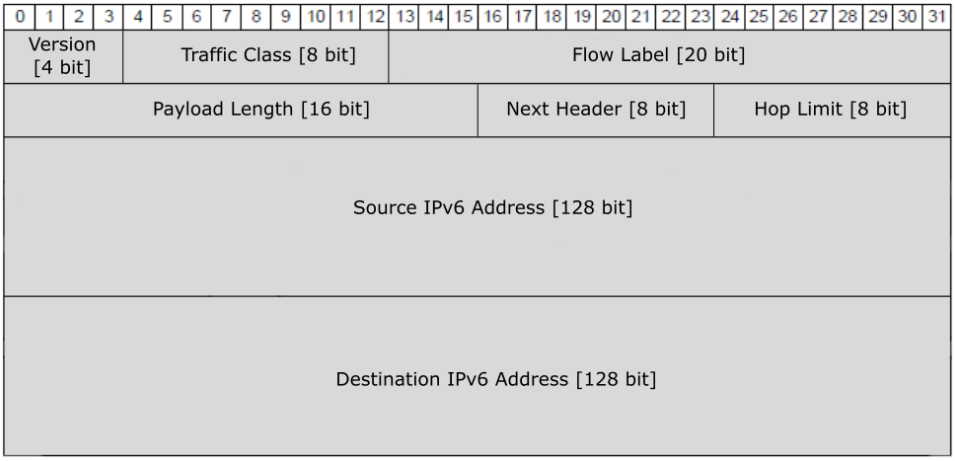


Figure 2.9: IPv6 packet structure

with the payload length (discarding the length of the header) and placing it after the Flow label field. Flow label introduces an opportunity to label a sequence of IPv6 packets. The Next header field specifies the type of the expanding header, e.i., the type of an encapsulated protocol. The last change to the IP header, excluding the address sizes, is the Hop limit field. This field replaces the TTL field, having practically the same function.

Despite its many advantages, IPv6 adoption is very slow, with many still relying on publically more utilized IPv4. This may be due to the fact that the transition from IPv4 to IPv6 can be complex and demanding. Thankfully, the automotive industry utilizes only the IPv6 protocol.

2.3 Transport protocols

Transport protocols belong to the fourth layer of the OSI model (Figure 2.1). The main purpose of these protocols is to provide services to the application. The most significant areas of these provided services are reliability, flow control, and muxing.

The best-known and most widely-used protocols are UDP (User Datagram Protocol) and TCP (Transmission Control Protocol). A more detailed overview of these protocols follows in upcoming sections.

2.3.1 UDP

User Datagram Protocol is a message-oriented transport protocol. Its main advantage and also disadvantage compared to other protocols is its simplicity.

UDP is connectionless, meaning it does not establish any connection stream to the receiver and treats a single packet, or a datagram, as a beginning and an end. This means each datagram must contain all the information for its delivery, usually enclosed in a header. Because of the simple design, UDP does not guarantee packet delivery, therefore, is branded as unreliable. The managing of reliability is left to the application.

Apart from the payload itself, UDP carries information about the source port and the destination port. These ports are 16-bit integers ranging from 0 to 65 535. Naturally, many of these ports are reserved or forbidden. The size of the UDP header and the payload is summarized in the length. The length field comprises of the size of the UDP header and the carried payload. To check for errors, a checksum field is added to the end of the header. The sender sums together all 16-bit segments of the packet, inverts all the bits of the sum, and places the resulting 16 bits inside the checksum field. The receiver computes the checksum similarly, but instead of inverting the bits, the receiver's checksum is added up to the sender's checksum. If the result contains any zeros, an error is detected, and the receiver discards the packet.

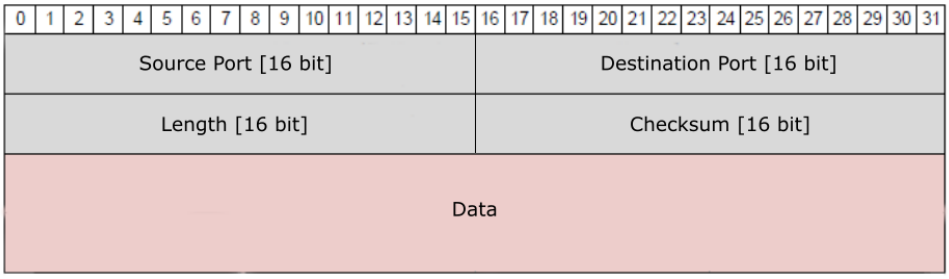


Figure 2.10: UDP datagram structure

In theory, the maximum length of the UDP payload is 65 527 bytes. However, the UDP packet is often encompassed within the IP protocol, which imposes a stricter limitation of 65 507 bytes (20 bytes of the IP header).

2.3.2 TCP

Unlike UDP, TCP (Transmission Control Protocol) is a connection-oriented protocol, e.i., the connection between the two units is established before the actual data transmission begins. TCP transport protocol is well-praised for its reliability, error-free operation, and data protection. Instead of sending data in a self-contained datagram like UDP, TCP creates a stream where numbered segments are transmitted. Each data segment is given a segment number, which is increased by one from the previous data segment. Thanks to this segment numbering, both sides keep track of all the segments, and in case any segment number is missing, a replacement is immediately sent.

The communication starts with a three-way handshake. In the first segment, the client sends a SYN (Synchronize Sequence Number) request. In this first message, the client selects an arbitrary number as a sequence number, and the acknowledgment number is set to zero. The server, after receiving the SYN request, responds with a SYN-ACK message. The sequence number is again arbitrary; however, the acknowledgment number is a previous sequence number iterated by one. This iteration is repeated for each upcoming message of a TCP stream. The last segment in a TCP handshake is the client's acknowledgment. The same iterating principle applies to this message. The structure of a TCP segment is more complex than its UDP counterpart, as seen in Figure 2.11.

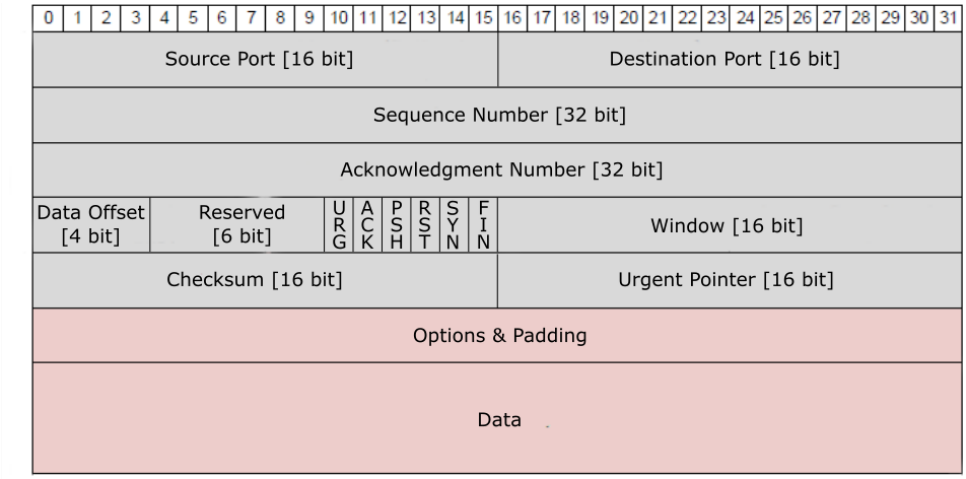


Figure 2.11: TCP segment structure

2.4 SOME/IP

SOME/IP (Scalable service-Oriented MiddlewarE over IP) [5] is a protocol used for communication between individual units in a distributed system. It is designed to enable secure, efficient, and reliable communication between devices of different sizes and operating systems that are connected over IP networks, such as Ethernet. Its most prominent utilization is in the road vehicles, where it can serve as a replacement for various traditional automotive protocols like CAN, MOST, LIN, or FlexRay. Two major disadvantages, or rather limitations, of these traditional protocols, are limited bandwidth and the inability to adjust to any change in the network's topology, software updates, or address changes. SOME/IP approaches communication from a different direction; instead of being signal-based like the protocols mentioned above, it is service-oriented [12]. The data exchange between the client and the server happens only when requested by the client or notified by the server

answering a pre-agreed subscription. This alone prevents any bandwidth wasting because the data is transmitted only when requested and where required.

As the name suggests, this protocol is a middleware. It sits upon the Ethernet and IP packet and forms a general foundation for a more specific and narrow protocol to deal with a particular application. One more protocol lies between the IP and SOME/IP, the transport protocol. Two options are available, TCP and UDP. Both have advantages and disadvantages, discussed in Section 2.3, but generally, the UDP protocol is preferred for its flexibility, as there are situations where TCP cannot be used. The data can be transmitted over UDP using unicast, multicast, or even broadcast.

Both transport protocols can include several SOME/IP messages. The payload in the TCP data stream can be of arbitrary size; this does, however, not apply to the UDP protocol. The maximum size of the segment is 1392 bytes. Payloads of greater size than the defined maximum must be fragmented into multiple packets. Handling these oversized packets is the responsibility of SOME/IP Transport Protocol (SOME/IP-TP). The fragmentation is signaled by a TP flag, set to one for all fragments apart from the last.

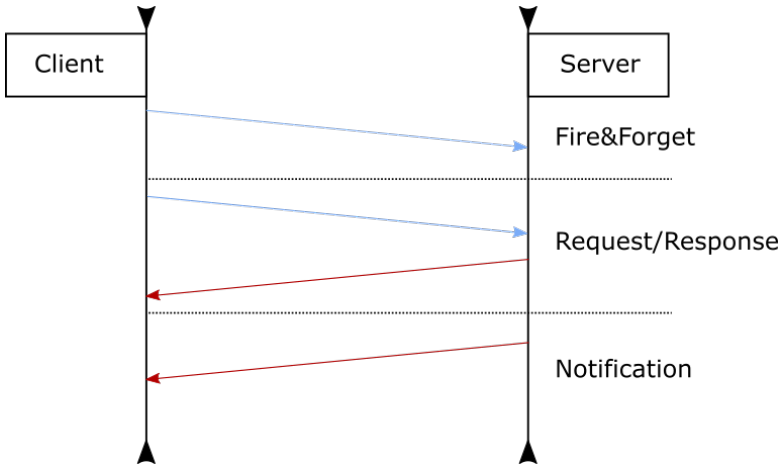


Figure 2.12: Standard SOME/IP message types

What makes the SOME/IP stack so flexible is its wide selection of message types. Remote procedure call (RPC) is a necessity in a distributed system, and SOME/IP is well equipped to handle it. Apart from standard message types, SOME/IP defines an extension solving the scalability of the network, the Service Discovery (SOME/IP-SD) [6]. The following standard messages are supported:

- **Fire&Forget** - the client sends a request with no response coming back from the server

- **Request/Response** - the client sends a request, and a response is sent back from the server
- **Notification** - a periodic message sent from the server to the client

2.4.1 SOME/IP-SD

As stated above, SOME/IP is a service-oriented middleware; for that, SOME/IP-SD offers many features. This extension allows the client to find services and subscribe to them dynamically.

- **Offer** - the client broadcasts a request to find a unit offering a service, and the server managing the particular service replies with an offer
- **Find** - the client establishes a subscription for a predefined time; the server periodically sends notifications
- **Subscribe** - the server broadcasts an offer containing information about the service, address, and port

2.4.2 SOME/IP Structure

The SOME/IP message consists of a header and a payload. The header contains all the necessary data for a correct interpretation, while the payload carries the actual data. The structure of the message is depicted in Figure 2.13. The 32-bit length field covers the whole payload and header, excluding only the Service ID, Sub ID, Method ID, and the Length field itself. The missing 17th bit in Figure 2.13 is a Sub ID, often included in the Method ID. The most notable fields of the header are analyzed below.

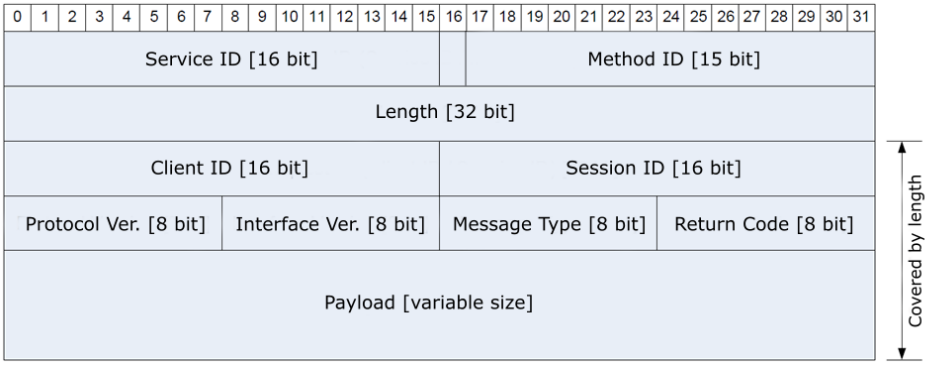


Figure 2.13: SOME/IP message structure

- **Service ID** - a 16-bit field responsible for identifying services
- **Sub ID** - a 1-bit field determining whether the message is a standard SOME/IP or SOME/IP-SD

Value	Description
0	SOME/IP
1	SOME/IP-SD

Table 2.4: Sub ID meaning

- **Method ID** - a 15-bit field identifying individual methods
- **Client ID** - a 16-bit field unique for each client communicating with a server
- **Session ID** - a 16-bit field describing a flow of communication
- **Message Type** - an 8-bit field determining the type of message according to Table 2.5

Basic Types		SOME/IP-TP	
Value	Description	Value	Description
0x00	Request	0x20	Request TP
0x01	Fire&Forget	0x21	Fire&Forget TP
0x02	Notification	0x22	Notification TP
0x40	Request ACK	0x60	Request ACK TP
0x41	Fire&Forget ACK	0x61	Fire&Forget ACK TP
0x42	Notification ACK	0x62	Notification ACK TP
0x80	Response	0xa0	Response TP
0x81	Negative response	0xa1	Negative response TP
0xc0	Response ACK	0xe0	Response ACK TP
0xc1	Negative response ACK	0xe1	Negative response ACK TP

Table 2.5: Individual message type codes

- **Request Code** - an 8-bit field indicating a status of an operation

Value	Description
0x00	OK
0x01	NOT OK
0x02	Unknown service
0x03	Unknown method
0x04	Not ready
0x05	Not reachable
0x06	Timeout
0x07	Wrong protocol version
0x08	Wrong interface version
0x09	Malformed message
0x0a	Wrong message type

Table 2.6: Return codes

2.4.3 SOME/IP-SD Structure

SOME/IP-SD message structure keeps the header of the standard SOME/IP message, even though it does not use all its features, and extends it with its own headers. The SOME/IP fields like service ID, method ID, or client ID remain constant for all service discovery messages and hold no added value. Moreover, the extension does not differentiate between message types, so 0x02 (notification) is always used.

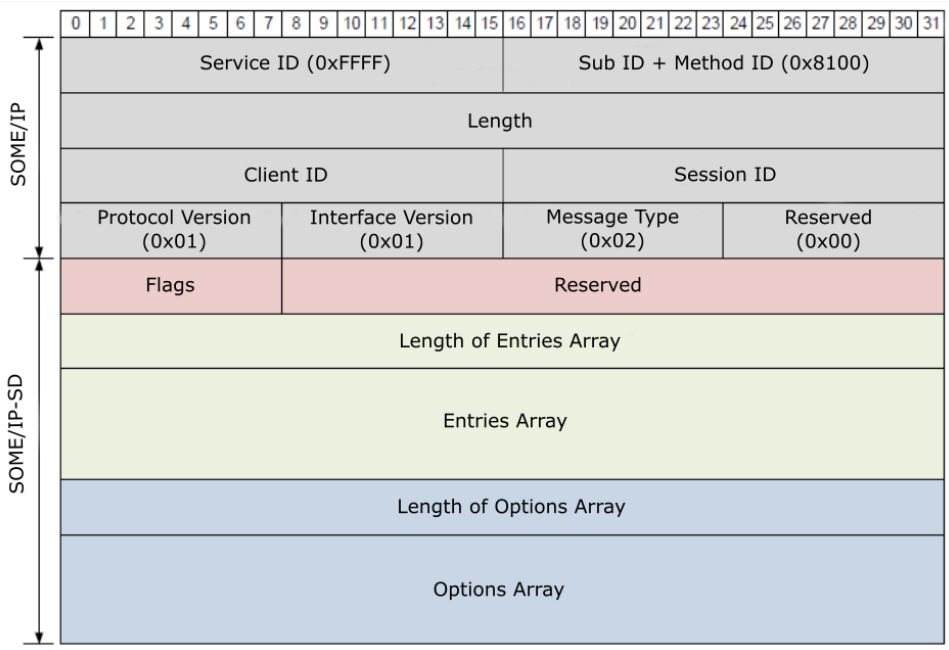


Figure 2.14: SOME/IP-SD message structure

After the standard header follows a 32-bit padding consisting of flags and reserved bits. The first bit of the padding holds a significant value; it is called a Reboot bit. This bit is set to a logical one when the unit has been rebooted so that all other units communicating with this one know of the reboot and adjust accordingly. The bit is flipped back to logical zero when the Session ID overflows. The added value of this extension is the inclusion of entries and options.

Entries

Entries are basic units of the SOME/IP-SD extension. Each message must contain at least one entry. Regarding structure, there are two types of entries, e.i., Service entries and Eventgroup entries. The former entries offer information about services, while the latter manages subscriptions. Each entry may have up to two sets of options associated with it. The structure of the former entry is depicted in Figure 2.15, the latter in Figure 2.16. The main difference in the message structure is in the last 32 bits. Service entry defines a 32-bit field containing the minor version. This is changed in the Eventgroup entry where the first 12 bits are reserved bits set to zero, followed by a 4-bit counter, and ending in a 16-bit eventgroup ID. The 9th bit in the reserved field is called Initial Data Request Flag and is set to one if the client requests the data immediately. All the shared fields are described below.

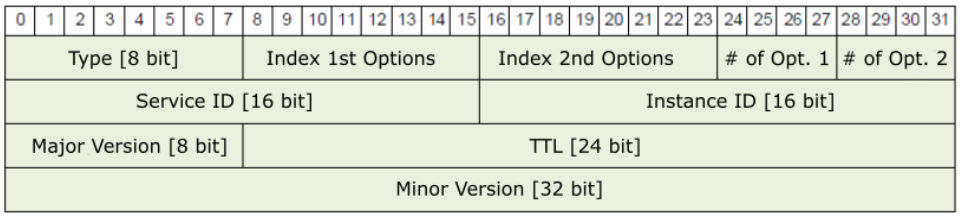


Figure 2.15: Structure of Service Entry

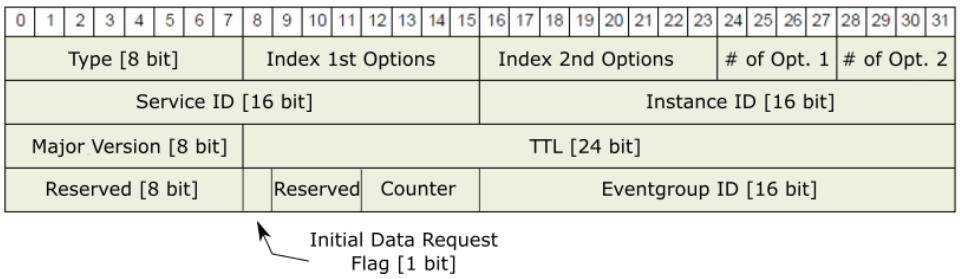


Figure 2.16: Structure of Eventgroup Entry

- **Type** - an 8-bit field describing a type of entry, the first three belonging to Service entry and the other two to Eventgroup entry
  - **Find Service (0x00)**
  - **Offer Service (0x01)**
  - **Stop Offer Service (0x02)**
  - **Subscribe (0x06)**
  - **Subscribe ACK (0x07)**
- **Index 1st options** - an 8-bit integer index of the first option associated with the entry in question
- **Index 2nd options** - an 8-bit integer index of the second option associated with the entry in question
- **# of opt 1** - an 8-bit integer defining the number of options in the first set.
- **# of opt 2** - an 8-bit integer defining the number of options in the second set.
- **Service ID** - a 16-bit field identifying a service
- **Instance ID** - a 16-bit field identifying an instance of a service
- **Major Version** - an 8-bit field containing the main version of a service
- **TTL** - (time to live) a 24-bit field describing the time of validity of the message in seconds

Options

Options are units providing additional information about an entry. All the options begin with the same three fields, e.i., length, type, and reserved.

- **Length** - a 16-bit field containing the option's length without the length and type fields
- **Type** - an 8-bit field describing the type of the option
  - **Configuration (0x01)** - carries an additional ASCII data
  - **Load Balancing (0x02)** - sets priorities for entries
  - **IPv4 Endpoint (0x04)** - carries information about an endpoint such as IPv4 address, transport protocol, and port
  - **IPv6 Endpoint (0x06)** - similar to IPv4 Endpoint but carrying the IPv6 address in a 128-bit field instead of the 32-bit IPv4 address field

- **IPv4 Multicast (0x14)** - carries information on where to send notifications with multicast; defines IPv4 address, transport protocol, and port
- **IPv6 Multicast (0x16)** - IPv6 version of IPv4 Multicast
- **IPv4 SD Endpoint (0x24)** - provides information on where to reach a SOME/IP-SD instance; defines IPv4 address, transport protocol, and port
- **IPv6 SD Endpoint (0x26)** - IPv6 version of IPv4 Multicast

- **Reserved** - an 8-bit reserve containing zeros

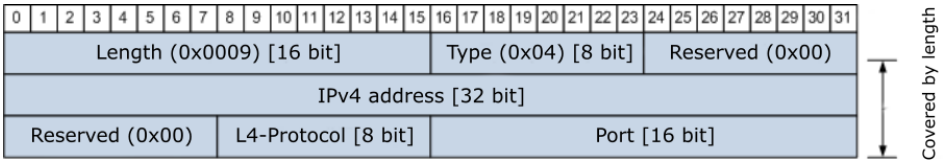


Figure 2.17: IPv4 Endpoint Option

2.4.4 SOME/IP Serialization

Serialization is the process of converting data structures into a binary format that can be transported as a payload of a SOME/IP message. This process is necessary because a computer typically represents data structures as complex objects with interdependent fields and methods. In contrast, network protocols like SOME/IP require a flat binary format for transmission.

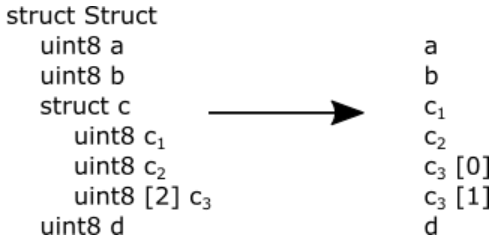


Figure 2.18: Data serialization

In SOME/IP, the data is aligned to an 8-bit format. All the base data types are described in Table 2.7. Structures made of these base data types are flattened consecutively, as shown in Figure 2.18. Fields with a dynamic length, such as a string, require a length definition fitted in front of the value.

This base serialization can be expanded into an extensible serialization. It envelops each element of a structure in a frame called TLV (tag/length/value). TLV frame is unique for each element in a particular structure; two different structures may share an identical TLV frame. The tag part consists of two parts; wire type and data ID. Wire type is a 4-bit field defining a default

Value	Description	Size
Boolean	TRUE/FALSE value	8
uint8	unsigned integer	8
uint16	unsigned integer	16
uint32	unsigned integer	32
uint64	unsigned integer	64
sint8	signed integer	8
sint16	signed integer	16
sint32	signed integer	32
sint64	signed integer	64
float32	floating point value	32
float64	floating point value	64

Table 2.7: Base data types

length of elements. Individual values are described in Table 2.8. The data ID is a unique part of the TLV tag; the value is encoded in 12 bits.

Value	Description
0x0	8-bit data base type
0x1	16-bit data base type
0x2	32-bit data base type
0x3	64-bit data base type
0x4	complex data type with length field of static size
0x5	complex data type with length field of 1 byte
0x6	complex data type with length field of 2 bytes
0x7	complex data type with length field of 4 bytes

Table 2.8: Wire types

Chapter 3

Implementation

With the theoretical background covered, a client capable of SOME/IP communication can be implemented. This client must strictly adhere to the AUTOSAR specifications [5, 6], meeting specified criteria and guidelines. This chapter describes the necessary components of the client, the implementation of the client itself, and the implementation of the virtual test server consecutively.

3.1 Packet definitions

Individual bits of a packet can only make sense with a middleware that is able to recognize the protocol and dissect it using the predefined set of rules. One such middleware is the Scapy project [8].

Scapy is a library written in Python that defines each layer of a packet as a class object. These classes are then layered on top of each other to construct a complete packet, as seen in Figure 3.1. With these class objects, creating a new packet or dissecting one received is effortless. Scapy has a wide variety of protocols already defined, but implementing a new one is pretty straightforward. Fortunately, SOME/IP is already implemented in an automotive contribution directory.

```
packet = Ethernet()/IP()/UDP()/SOMEIP()
```

Figure 3.1: Scapy packet creation pseudocode, layering SOME/IP on top of UDP, IP and Ethernet.

It is important to keep in mind that Scapy is only responsible for dissecting the header of a packet, sending the payload for additional processing. This

means that the encoding and decoding of raw data is left to the user, and so is left to be implemented.

However, Scapy can do much more than define layers. Scapy offers tools to send and receive the packet both on the second and the third layer of the OSI model, thus serving as a backbone for a whole communication framework.

3.2 VW Resource Protocol

With a middleware for creating SOME/IP packet developed, we have to focus on the data it is to transport. It is a common practice to define custom data structures, usually a vector containing simple datatypes such as integers, floats, or booleans. These structure definitions are generally kept a secret within a company using them. As we work with Volkswagen Group hardware, we are interested in their definitions, which are obviously not accessible to the public. Luckily, we were kindly supplied with .xml and .arxml files containing the necessary descriptions of selected services for testing.

A parser was developed to load data from these description files and transform them into a structure that can be used to encode and decode data by Volkswagen Group standards. This parser uses the lxml library [11] to load the .xml file as a tree, enabling a fast and easy search. The output of this parser is a set of machine-generated objects working as templates that allows us to transform values into bytes and vice versa. Moreover, these objects give meaning to individual values. As it is not allowed to share data from these supplied files, we use made-up values for the visualization in Figure 3.2. On the top of the tree are services; each of these services has a few methods that can be called, each containing a vector of values.

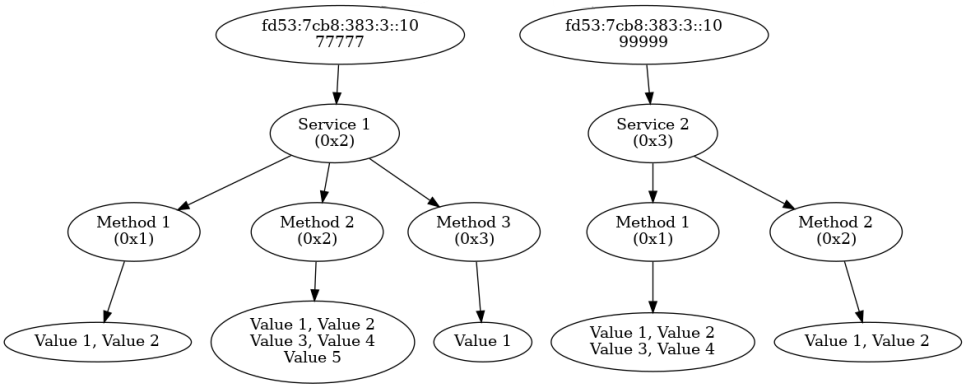


Figure 3.2: Visualization of the parser's structure

For convenience, a second parser was implemented, taking definitions from files intended for Wireshark use. The structure of these files is entirely different from the .xml files, thus needing a completely new approach. As the structure resembles the text file, reading line by line does the trick. This parser works from the ground up, starting with data codings and working its way to the services themselves. The output of this parser corresponds to the output of the first solution, so its utilization remains the same.

The output of these parsers is a set of objects representing individual services. Each service has at least one attached object representing a method. By calling on this object, a set of bytes can be either created or decoded, depending on the use case.

3.3 Client

The client is written as a multi-threaded Python program, controlled by both to the graphical user interface and the console. The whole project is written in Python 3.8.10. The client consists of four threads that are run only once and are in a state of an endless loop, e.i., main, asynchronous listener, offer, and console; these are called primary threads. The rest of the threads can be divided into three types, e.i., find, send, and subscribe; these are the secondary threads. The aforementioned threads are started to manage a specific task, and multiple instances can run simultaneously. Each thread of either group is fitted with locks to prevent collisions, as they access the same memory locations.

3.3.1 Main thread

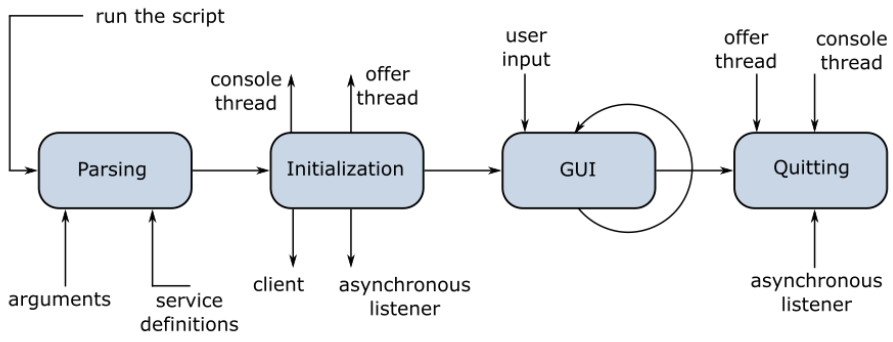


Figure 3.3: Client main thread flowchart

The main thread is the first thread started, thus, is responsible for the correct initialization of the whole program. The function of the main thread

can be divided into four phases, as is depicted in Figure 3.3, e.i., parsing, initialization, GUI, and quitting.

Parsing

Parsing is a process of analyzing text files or input to decode useful data. The data to parse comes from the bash script the user called to start the client Python program. The user is expected to supply the machine's IPv6 address and the port for the communication and write them in the bash script. The same must be done with the broadcast IP address and port. Additionally, the user is expected to supply the client with the communication interface, in our case, the name of the Ethernet interface. The last set of arguments, the IP address and port of the test server, is optional. Filling them automatically runs the program in debug mode, where the communication is expected to be only with the test server. Leaving them blank runs the program in normal mode.

```
python3 client.py --ipc $IPC --pc $PORTC --ipb $IPB --pb
    ↪ $PORTB --ic $IFACE
```

Figure 3.4: An example of a command to run the client in normal mode

An example of a command with arguments running the program in normal mode is presented in Figure 3.4. The arguments given to the client are displayed in the following order:

- **ipc** - the client's IP address
- **pc** - the client's port number
- **pb** - the broadcasting IP address
- **ipb** - the broadcasting port number
- **ic** - the interface

The second type of data the program needs to parse is the services from the available definitions. This process is described in Section 3.2. The program initializes the Parser class instance and exports the services sorted in an alphabetical order. Once the set of services is exported, the Parser class instance is deleted from the memory.

Initialization

Once the data is parsed, the program is ready to initialize the client class instance. Its parameters are the IPv6 addresses, ports, and available services. The client, at this stage, prepares the threading locks and the SOME/IP parameters such as session ID, message sending timeout, or whether the client is supposed to wait for the acknowledgments from other units. With the client class instance set, the program creates the rest of the primary threads and supplies them with the appropriate functions.

Algorithm 1 Start New Thread

1: $e = Event()$ ▷ create threading Event
2: $t = Thread(func, e)$ ▷ create the thread
3: $t.daemon = True$ ▷ set threat to daemon
4: $lock.acquire()$ ▷ acquire the thread lock
5: $th =$ thread record ▷ contains thread, event, ID, and SD flag
6: $threads.append(th)$ ▷ all threads are recorded in the client
7: $lock.release()$ ▷ release thread lock
8: $t.start()$ ▷ start the thread

GUI

When the main thread enters the GUI phase, it initializes the GUI class instance, imports the service templates, sets callbacks to the buttons, and enters the infinite loop until the quit button is pressed. During GUI initialization, individual elements of the window are set, as well as the resolution and color theme. When the service templates are received, they are displayed in the tab called "Available Services" and each template is given a callback function to display available actions, which in turn get the callback to show methods. In the case of the subscription, the method is replaced by the eventgroup. The window frame is cleared before the individual items are displayed, and the previously selected item's background is darkened. Finally, each method, or eventgroup, gains a callback to display individual data fields. The way the data fields are displayed depends on the selected action.

- **Get** - the form to fill individual values is disabled
- **Set** - the form to fill individual values is enabled, and at least one value is expected to be filled
- **Notification** - the form to fill individual values is enabled, and at least one value is expected to be filled
- **Subscribe** - the form to fill individual values is disabled

A similar process takes place in the case the user wishes to display additional information about a received message. It is treated as a "Get" action. The individual templates are determined using the message header, and their respective labels are darkened.

Algorithm 2 Send message callback

```
1: get service
2: if action is "Subscribe" then
3:     ttl = 5
4:     get eventgroup
5:     session_id ± 1
6:     start the subscription thread
7: else
8:     get method
9:     if action is "Get" then
10:        response = True
11:    else
12:        response = False
13:        get values
14:    end if
15:    session_id ± 1
16:    start the send message thread
17: end if
```

The GUI interface has two main buttons: the send and quit buttons. The former's callback collects all the selected data, increments the session ID, and starts a send message thread. The latter moves the program to the quitting phase.

The last two functions defined by the GUI class instance log events and offers. In both cases, the timestamp in the form of hours, minutes, and seconds is displayed, followed by the event or offer information. The log distinguishes between two types of records, the clickable and the unclickable. To the former, a callback is added to display additional information when clicked on. The most notable difference between logging events and offers is the way the refreshing of the window frame is handled. The log simply adds a record to the end; however, the offer window is cleared and refreshed each time it is called, as each record has a lifespan.

Quitting

Upon the raising of the quit flag, the main thread gracefully ends all other running threads, logs this into the console, and ends the program. In case the program is run in debug mode, the client sends a message to the test server signaling it to end. The client does not wait for any response from the server and shuts down.

3.3.2 Asynchronous listener

The asynchronous listener is a thread monitoring ports for any incoming packets. These packets are filtered according to the transport protocol used, permitting only UDP and TCP protocols. For each received packet, a routine is called to dissect the packet and react appropriately. First, the packet is checked to see if it is indeed a SOME/IP packet. A SOME/IP message type can be determined by its message type field value, as in Table 2.5. According to the message type, the received message can be categorized into four groups, e.i., ACK, Response, Notification, and SD.

- **ACK** - a thread with a corresponding session ID is signaled that the acknowledgment message has been received
- **Response** - a message is first checked for errors, then the individual values in the payload are processed and displayed in the console as well as in the GUI log
- **Notification** - in case of notification, the individual values in the payload are processed and displayed in the console, and the GUI log
- **SD** - an SD message can be either a find service, offer service, or subscribe to a service message; all is well documented in the console, and the GUI log

3.3.3 Offer thread

The offer thread is responsible for keeping track of all the offers and removing the expired ones. It first acquires a lock to access the offers, gets the current timestamp, and removes the expired offers. Each offer has its own class instance with the following variables:

- **Name** - the name of the offered service
- **Address** - the IPv6 address of the ECU providing offered service
- **Port** - the port of the ECU providing offered service
- **Time** - the timestamp of expiration of the offer

The main concern of this thread is the 'Time' variable. It depicts the time when the offer becomes expired. This variable is then compared to the current time value, popped from the list if the current time value is greater than the saved variable. This check is repeated four times a second; however, this parameter can be easily tuned. The remaining offers are rerendered in the GUI offer log.

Algorithm 3 Offer Thread

```
1: while not Event() is set do                          ▷ external shut down
2:     lock.acquire()                                   ▷ acquire offer lock
3:     time = time()                                    ▷ get actual time
4:     for each offer in offers do
5:         if offer.time ≤ time then
6:             offers.pop(offer)                        ▷ remove offer from offers
7:         end if
8:     end for
9:     lock.release()                                   ▷ release lock
10:    display offers in GUI
11:    wait()                                           ▷ sleep for specified time
12: end while
```

3.3.4 Console thread

The console thread is in a state of an endless loop, waiting for the user input. As with any other thread in this implementation, the loop can be exited by raising a threading event from a main loop. The control keys are shown in Table 3.1. Two operations, namely sending messages and subscriptions, deserve further analysis.

Algorithm 4 Send message via console

```
Require: input = "r" or input = "s"
1: session_id ± 1
2: logging = False
3: display services
4: service = input()                                    ▷ wait for user input
5: display methods
6: method = input()                                     ▷ wait for user input
7: if "Set" then
8:     while True do
9:         display fields
10:        field = input()                              ▷ wait for user input
11:        if field is empty then
12:            break                                    ▷ ending the input
13:        end if
14:        value = input()                              ▷ wait for user input
15:        set value
16:        display values
17:    end while
18: end if
19: start send message thread
```

- To send a message, the input is expected to be 's' or 'r' (according to Table 3.1), the former designating the fire&forget message, the latter the request/response transaction. When either of these two inputs is detected, the session ID is incremented and logging is disabled. Available services are displayed, each designated with an index. The thread then waits for the user to input the chosen index. The index is checked, and the service template is obtained. The same process is repeated with the method. If the original input is 's', individual fields are displayed, and the thread waits for the user's input for the index of the field and a value. After each filling, the message is displayed again with the new values. At least one field must be filled before sending the message. The payload is then transformed into bytes, and logging is enabled. Finally, the start message thread is started.

- The subscribe procedure is designated by the input 'p' (Table 3.1). Similarly to the standard SOME/IP message sending, the console stops logging and waits for the user's additional input to determine the service and method. Again the session ID is incremented, and a subscription thread is started, after which the logging is enabled.

3.3.5 Find thread

When the IPv6 address or port of the ECU providing a requested service is not known, a find service thread is started. A service discovery message of type service entry is broadcasted into the network. The unit in question is expected to immediately broadcast the offer service message. Once the parameters are known, they are returned to the calling function, and the thread ends.

Algorithm 5 Find Thread

```
1: get the service template
2: acquire offer_lock
3: scan offers for address and port
4: release offer_lock
5: if found address and port then
6:     return address and port
7: end if
8: construct SOME/IP-SD Find Service entry message
9: broadcast the message into the network
10: wait for offer 5 times for 0.1 seconds
11: if received offer then return address and port
12: else
13:     log failure
14:     return None
15: end if
```

3.3.6 Send message thread

In order to send a message, a dedicated thread is created so as not to limit the client's responsiveness. First, the service record is checked to determine whether the destination address and port are known. In case the data about the unit is missing, all current offers are checked. If these parameters are still unknown, a find thread is started to send a find service inquiry, as described in Section 3.3.5. If the address and the port still remain unknown, the failure is logged in the console and the GUI log, and the send attempt is aborted.

Once all the necessary information is known, a SOME/IP message is constructed and filled with the supplied parameters from the user input. The message type is set to request in case of a "Get" action and a fire&forget if the selected action is "Set". The payload is checked for the need for fragmentation, and the SOME/IP-TP method is used if required. The created SOME/IP packet with the payload is embedded into the transporting protocols (Ethernet, IP, and UDP). The message is sent, and the action is recorded in the console and the GUI log. The thread is ended if the acknowledgment message is not expected.

Algorithm 6 Send Message Thread

```
1:  get service address and port
2:  if fragmentation needed then
3:      TP flag ← 1
4:  end if
5:  create SOME/IP header
6:  if payload then
7:      if TP flag then
8:          for each fragment without the last do
9:              send fragment
10:         end for
11:         TP flag ← 0
12:     end if
13: end if
14: send message                          ▷ or last fragment if TP
15: if wait for ACK then
16:     wait for ACK
17: end if
18: acquire thread lock
19: remove itself from the record
20: release lock
21: end thread
```

If the acknowledgment message is expected, the message send thread is set to the waiting state, watching for the signal from the asynchronous listener.

If the acknowledgment message is received within a specified timeout, the operation is successful. The end is then the same as in the non-waiting scenario.

3.3.7 Subscribe thread

The subscribe thread is, at its core, very similar to the send message thread. Again the first action is to find out the address and the port of the unit providing the wanted subscription. An appropriate service template is looked up and used to create the SOME/IP-SD packet embedded in the SOME/IP packet. The IPv6 option is added to the SOME/IP-SD packet with the information about the client. The message is sent, and the thread is set to the waiting state, waiting for the eventgroup acknowledgment message. The ACK signaling is identical to the one implemented in the send message thread. The result is displayed in the console and the GUI log, after which the thread ends.

3.4 Test server

A test server was implemented to enable testing and tuning the client without the need for real hardware. The test server has been written in such a fashion that extending it and implementing new features is easy. The program works as a typical server unit, listening to the ports and providing responses to the individual clients, in this case, only one client. Similarly to the client, the server is written as a multi-threaded Python program. The main thread can be divided into three phases: parsing, initialization, and sniffing.

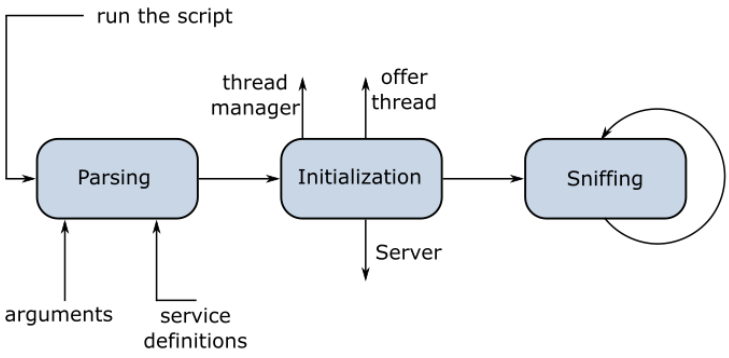


Figure 3.5: Server main thread flowchart

Parsing

In a parsing phase, the server first gets the arguments from the command line, e.i., the client's IP address and port as well as its own IP address and port. The default ports for client-server communication are 30480 for the server and 30490 for the client. The second data to parse is the set of service definitions. This process is identical to the service definitions parsing on the client side, described in Section 3.2.

Initialization

The initialization phase contains the initialization of the Server class instance and two primary threads. The Server class instance takes as an argument all the data gained from the parsing phase. It prepares all the locks for the threads and sets the values of the parameters such as the rate of supplying notifications, rate of sending offers, and flag, deciding whether an acknowledgment message is expected.

The first primary thread is the thread manager. It keeps track of each running thread and deletes the record of those that have ended. It runs its checking loop each second; however, this can be tuned to the user's needs.

The second primary thread is the offer thread. This thread offers the selected service to the client. Multiple instances of this thread can be run, by default, only one service is offered, meaning a single instance is started. The rate of sending offers can be tuned in the Server class instance, however, all offer threads must run at the same rate. When sending an offer, a SOME/IP packet with an incremented session ID is created, in which a SOME/IP-SD offer service entry is embedded. A default time-to-live value is five seconds. An IPv6 endpoint option is added to the SOME/IP-SD frame containing the server's IPv6 address and port. Finally, the message is sent to the client.

Sniffing

The primary purpose of the test server is to listen to the messages transmitted by the client, check them, and provide replies. First, the server checks if the message contains a SOME/IP header. Using this header, more information about the message can be found out. The presence of service discovery extension can be determined using the Sub ID bit. In case the service discovery extension is present, the message is dismantled into individual entries, which are processed one by one. The server is expecting only two types of SOME/IP-SD messages, the find service entry and subscribe eventgroup entry.

Algorithm 7 Subscription Thread

```
1: sleep_time = 60/rpm                    ▷ rpm is given from Server initialization
2: max_c = ttl/sleep_time                 ▷ ttl is received from Client
3: c = 0
4: get the method template
5: while c < max_c do
6:     fill values randomly
7:     construct SOME/IP notification message
8:     send notification
9:     count ± 1
10:    wait(sleep_time)
11: end while
```

Algorithm 8 Server Sniffing

```
1: while True do
2:     sniff 1 packet
3:     isolate SOME/IP frame
4:     check SUB ID
5:     if SD then
6:         for every entry do
7:             get corresponding option
8:             if entry type is Find Service then
9:                 send offer
10:            else if entry type is Subscribe Eventgroup then
11:                if service is offered then
12:                    send Eventgroup ACK
13:                    start subscribe thread
14:                end if
15:            end if
16:        end for
17:    else
18:        send ACK
19:        check for TP flag
20:        decode payload
21:        if action is "Get" then
22:            get method template
23:            fill values randomly
24:            send response
25:        end if
26:    end if
27: end while
```

- **Find service** - the service ID is checked, and if available, an offer is sent to the client
- **Subscribe eventgroup** - the service ID is checked, and if available, the server responds with a subscribe eventgroup acknowledgment message and starts the subscription thread

In the absence of the service discovery extension, an acknowledgment message is returned to the client. The message is checked for the TP flag, awaiting the following fragment if set. The test server works only for the "Get" action message, e.i., the fire&forget mode. If the message contains any payload, it is decoded and checked for test commands from the client. Standard "Get" action SOME/IP messages do not contain any payload; thus, it is easy to separate the test messages from SOME/IP ones. For standard SOME/IP messages, the service and the method are determined, and the appropriate values are filled with random numbers corresponding to their codings. The test commands are in the form of a string encoded using the UTF-8 standard [17]. The following three commands were implemented:

- **Start Offer Thread x** - a new offer thread is started, offering a service with ID = x
- **Stop Offer Thread** - all currently active offer threads are ended
- **Quit Server** - all threads are ended gracefully, and the server shuts down

3.5 User interface

The client is controlled via the built-in graphical user interface. Controllability via the console has been retained, but usage of the graphical user interface is preferred.

3.5.1 Graphical user interface control

The graphical user interface is divided into five main sections.

The first section, located in the upper left corner, shows the list of available services which have been received from the parser. By clicking on a service, available actions are shown to the right. Depending on the chosen service, up to four actions appear: Get, Set, Notification, and Subscribe. After choosing an action, more information is displayed in the information section on the right. Here the user is expected to select the desired method. If chosen action is get or subscribe, the send button is enabled, and the message can be sent.

Otherwise, the method's data is displayed in a form in the right part of the information section. After at least one value has been set, the send button is enabled, and the message can be sent.

Above the information section is the section advertising received offers. Each log consists of the remaining time and the service name. The remaining time is decremented every 0.25 seconds so to give a rough idea until when the offer is valid.

In the bottom left corner is the log. The log prints everything that happens to the client, such as confirmation that the message is sent successfully or a notification that the client received a message. Messages containing some data are clickable, and additional information about the message is presented in the information section. At the moment, only send button and quit button are situated here. The send button takes the data from the form above and signals the client to send the message. The quit button shuts down all threads running for the client and ends gracefully. The layout of the graphical user interface can be seen in Figure 3.6.

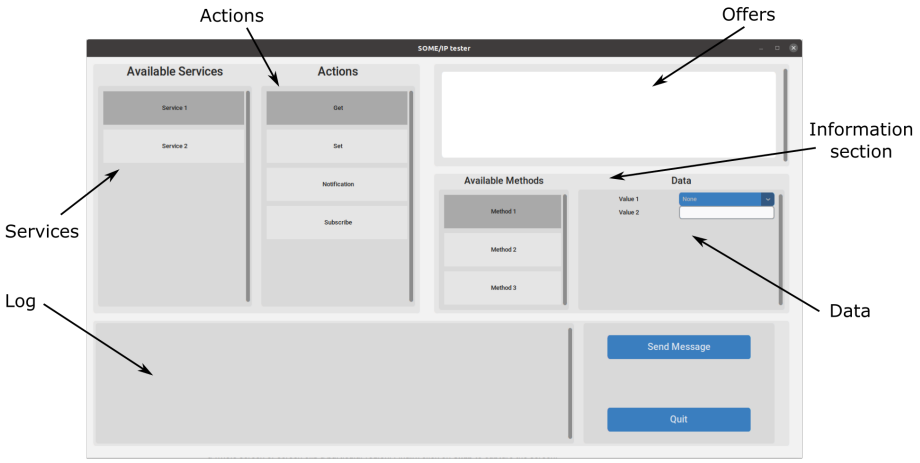


Figure 3.6: GUI layout

In order to send SOME/IP messages from the graphical user interface, a service has to be chosen in the service section. By clicking on the service, the label darkens, and the selection of methods appears in the information section. Select a method, and its data content will be visible in the right part of the information section. Fill in the form; at least one value has to be filled in. When the form is filled, press the button with the label "Send" to signal the client to send the message.

To subscribe to a service, simply choose a service and method with the subscribe action and click the send button. If the subscription was successful, a confirmation message appears in the log. After that, the log periodically displays the subscribed service. Again this message is clickable for additional information.

3.5.2 Console control

For convenience purposes, the console control has been retained. This approach's main drawback is its non-flexibility and difficult user-friendliness. The control keys are laid-out in Table 3.1.

Commands	
Input	**Description**
h	Print the help
s	Send a fire&forget message not demanding a response
r	Send a message demanding a response
a	Display all current offers
p	Subscribe to a service
u	Unsubscribe from a service
q	Quit a client
Testing commands	
Input	**Description**
m	Send a magic cookie
o	Signal a test server to start a offer thread
t	Signal a test server to stop all offer threads
f	Ask a test server if it supports a service
e	Quit a client as well as test server

Table 3.1: Description of individual console commands.

To send a message, it is required to press the corresponding key on the keyboard, choose a service and a method, and fill the fields with values. A more detailed description is provided in Section 3.3.4. The process of sending a SOME/IP message is captured in Figure 3.7.

What a graphical user interface does not possess is a testing suite for a test server. By pressing the corresponding keys, the client can control the test server and its behavior. Operations such as creating and ending offer threads, finding certain services, or sending a magic cookie are exclusive to the console control and the testing suite. To learn more about the testing commands, see Section 3.4.

```
----- Select a service from the list -----

0: Service 1
1: Service 2
1

----- Select a method from the list -----

0: Method 1
1: Method 2
1

----- Select a field from the list to set -----

0: Value 1
1: Value 2
1
Input a value for field Value 2
11
[None, 11]

----- Set another value or hit enter to send -----

0: Value 1
1: Value 2

INFO: Sending service Service 2, method Method 2 with
data [None,11]
```

Figure 3.7: Imitation of the message sending operation via the console, sending service Service 2, method Method 2 with data Value 2 = 11

Chapter 4

Testing

At first, the implemented client was tested with a virtual server, described in Section 3.4. After the validation of its functionality, a testing HIL (Hardware-in-the-Loop) was built to verify its performance on real hardware, testing the functionality of the client as well as the behavior of the units.

4.1 Virtual server

The tests with the virtual server aim to prove that the SOME/IP protocol was well implemented and functions according to the specifications. Apart from the functionality validation, the client's parameters can be measured, primarily the time it takes to complete specific tasks. These tests were conducted on a single portable computer running Ubuntu 20.04 with a seventh-generation Intel i7 processor, which may influence the measurement of the client's parameters.

But first, the client was tested for functionality defined by AUTOSAR specifications. The client was tested on how it handles regular SOME/IP, SOME/IP-TP, SOME/IP-SD, and overall communication. The client passed all the necessary tasks, clearing the way for testing the client's features and qualities.

To test the client's performance, the operation time was measured for its critical tasks. For objectivity, each measurement was repeated 25 times and averaged. The test results are displayed in Table 4.1; all presented values are in milliseconds.

The first performed action was to send a "Get" message to the server without knowing the unit's address. The timer started with a push of a button in the GUI and ended the moment the message was sent. The measured values varied substantially, with a standard deviancy of nearly 25 milliseconds (coefficient of variation over 13 %).

The second action was acquiring a subscription. The conditions were identical to the prior operation; still, the subscription process managed to overdo the previous action. The deviancy increased to approximately 31 milliseconds (coefficient of variation of 16.3 %).

Description	Average	Maximum	Minimum	Deviancy
send "Get" message	184.092	225.979	145.416	24.95
subscribe to a service	190.111	234.097	144.852	30.974
find service	146.056	159.297	129.641	10.103
parsing phase	342.724	370.178	319.1	17.45
initialization phase	1.53	1.944	1.204	0.231

Table 4.1: Measured times for virtual test, displayed in milliseconds

The surprise of this test came in a standard deviancy calculation for the find service procedure. It was expected that the most significant contributor to the overall deviancy would be the find service thread, as it must wait for the response. This thread, however, contributed to the deviancy only marginally, with just 10.7 milliseconds (coefficient of variation under 7 %). Without the find address procedure, the send message and subscribe actions had coefficients of variation of 25.6 % and 39.1 %, respectively.

The parsing phase timing was conducted with the second parser (parsing from Wireshark definitions), as they include a wider selection of services. The average time of 343 milliseconds is understandable, considering the data is parsed from over 18 thousand lines spread over ten files.

4.2 HIL

When the theoretical functionality was confirmed, it was time to move to more practical tests. For this purpose, a functioning HIL was assembled. These units were originally intended to be used in Enyaq, Škoda's first electric SUV (sport utility vehicle). The units were connected together according to internal schematics. A broader schematic is shown in Figure 4.2. The full list of individual units is presented here:

- **ICAS1** - (In Car Application Server 1), a primary server offering most of the services
- **ICAS3** - (In Car Application Server 3), a secondary server offering some additional services
- **OCU** - (Online Connectivity Unit), a unit in charge of wireless communication, such as LTE networks

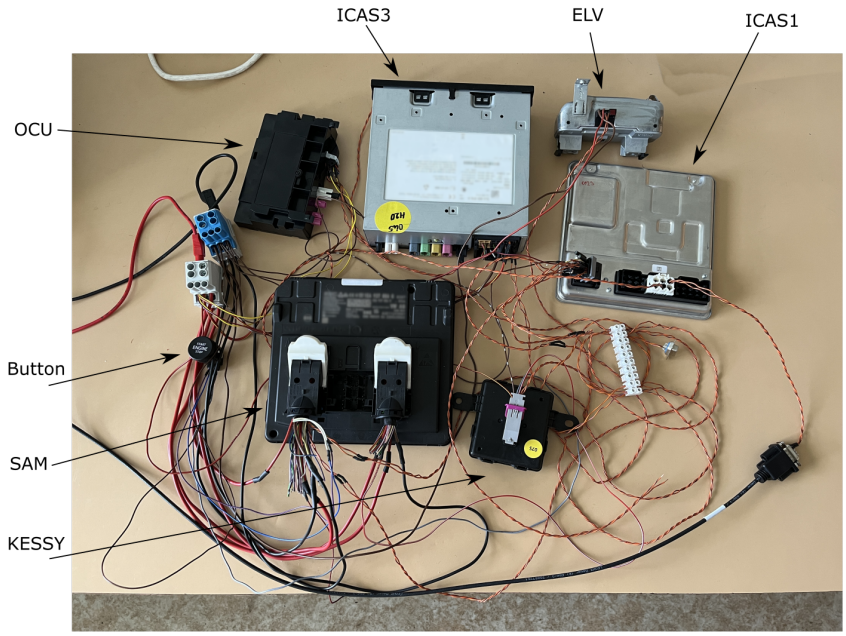


Figure 4.1: Photo of the connected HIL

- **SAM** - (Signal Acquisition Module), a unit receiving data from sensors and controllers
- **ELV** - (Electronic Steering Lock), a unit responsible for locking and unlocking the steering wheel
- **KESSY** - (Keyless Entry Start and Exit System), a unit enabling keyless access and starting
- **Button** - a standard button often found next to the steering wheel, used to start the car without the need for an ignition key

The units were connected via the standard wire, colored black in Figure 4.2. The coloring of additional connections depends on the interface used. The twisted pair cable transmitting the Ethernet protocol is colored blue, while the twisted pair cable used for CAN (Controller Area Network) is colored red. The power supply cables and ground cables were omitted from the schematic. The Ethernet line between ICAS1 and ICAS3 was split to connect the computer running the client using a switch. The HIL was supplied with 12 volts and an average current consumption of 1.6 ampers from the Agilent E3633A power supply.

In order to infiltrate the private network present in the HIL, the client has to imitate and replace one particular unit. As we split the Ethernet between ICAS1 and ICAS3, we chose to disconnect the ICAS3 unit and replace it with

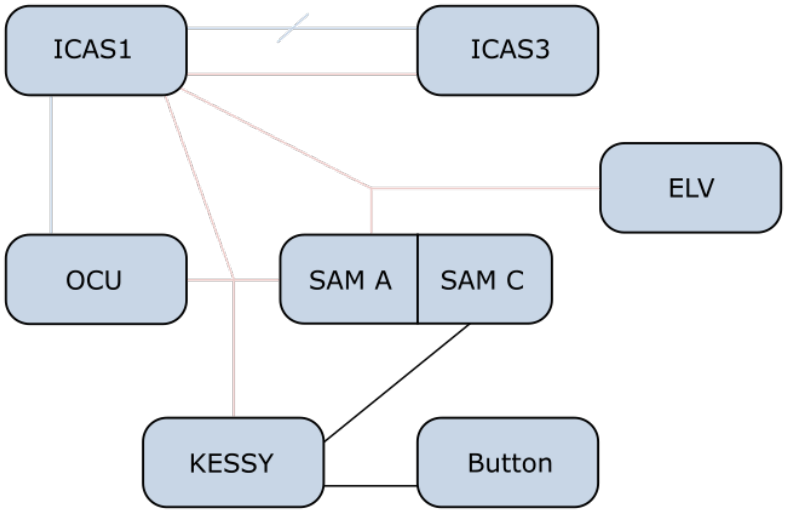


Figure 4.2: HIL schematics

our client. ICAS1 was manually added to the neighbour table. Finally, the switch was configured to add a VLAN tag to each packet coming from the computer running the client. The correctness of the computer's connection was verified by carrying out a ping command sent to the ICAS1 unit. The average time measured between sending and receiving the ping was 0.495 ms.

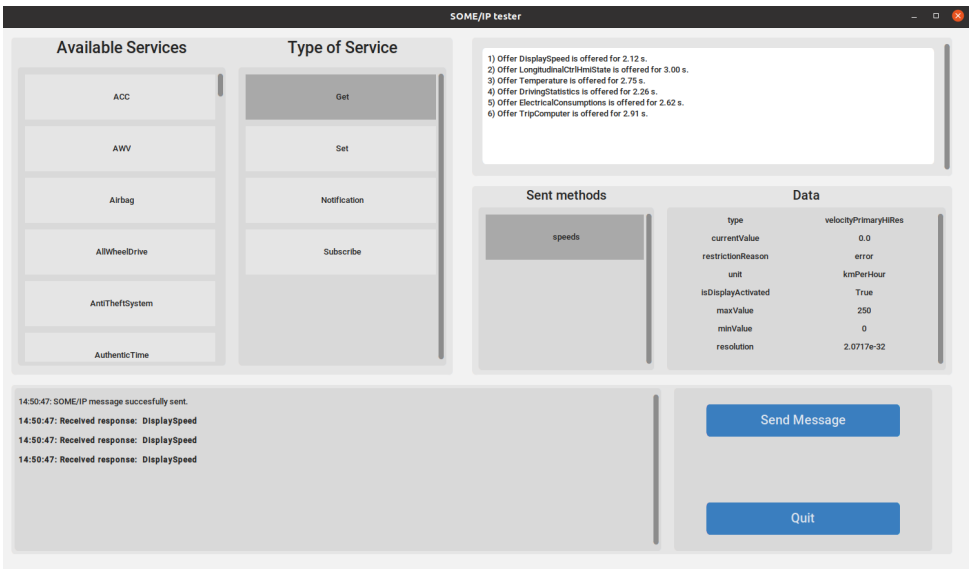


Figure 4.3: Image of a GUI after sending a "Get" request

With the computer successfully connected to the network, the client was supplied the necessary arguments and started. Immediately over fifty offers were displayed in the GUI offer tab. Sending a "Get" message to the unit

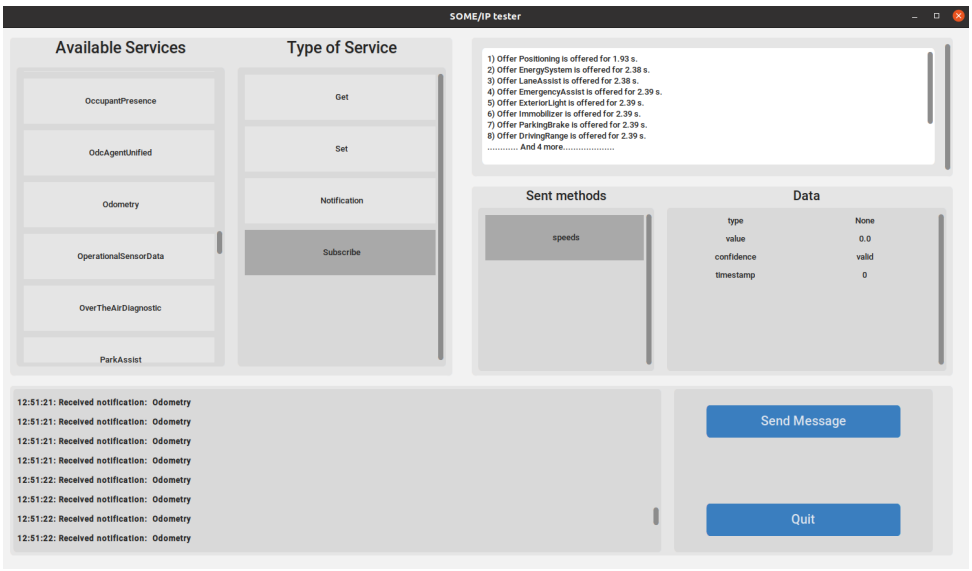


Figure 4.4: Image of a GUI receiving notifications from a subscription request

correctly returns the default values, as we do not have any sensors and actuators connected. The time between sending and receiving the message, measured for ten different services, was 8.22 ms. Little disappointingly, setting a value results in a successful response; however, requesting them after returns again the default values. It is unclear whether the values were set and immediately replaced by default values or whether the values were never changed. The next step was to try the subscription. After realizing that all the service discovery messages must be sent from the broadcast port, a successful subscription acknowledgment was received. The average time of waiting for this acknowledgment out of 10 measurements was 0.256 ms. This average value is substantially smaller than the one measured for standard SOME/IP messages. This makes sense, as, in this case, the unit does not need to get the values from the memory and search templates for the particular service. The average time between the subscription request and the first notification is much closer to the standard request message, 10.063 ms. The delay between each notification varies from service to service.

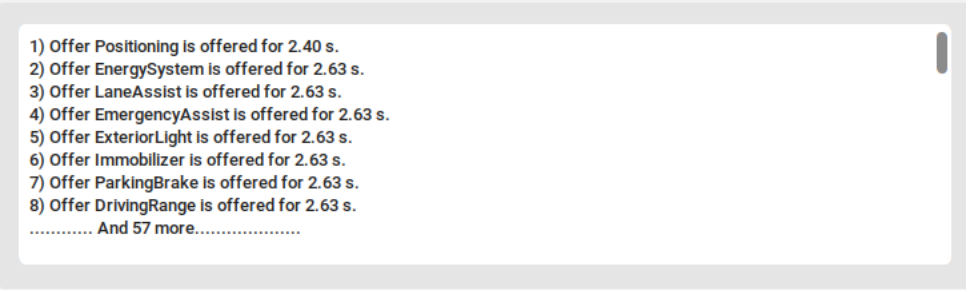


Figure 4.5: Zoomed in offer log

Next test aimed to verify the error code implementation inside the unit.

(a) : Log **(b) :** Data

Figure 4.6: Zoomed GUI log and information section

The aim was to malform the packet in some way to trigger the error response. By malforming the packet, we were not able to observe any unexpected behavior, and the unit simply returned an error code. It must be added that this test was only superficial, not proving any invincibility of this protocol. This remains to be explored in future studies.

Even though the ICAS1 offers more than fifty services, the majority of them do not return any payload, which probably means they are yet to be implemented by the manufacturer.

Chapter 5

Conclusion

In this thesis, all the protocols needed for SOME/IP communication were described in detail. All the findings were utilized in writing a SOME/IP client capable of standard SOME/IP communication, as well as the use of the service discovery extension. A client using SOME/IP according to the specified descriptions was implemented, together with a parser used to get information about individual services. A complimentary virtual test server was implemented for fundamental testing and debugging. The final SOME/IP client was run on a real HIL and subjected to a couple of basic tests. It is unfortunate that the units were not nearly as prepared for a full-scale SOME/IP communication as the client allows.

Because of its multi-threaded foundations secured by locks, the client is fast, robust, and equipped to handle an extensive network. During testing, the client worked as expected, prepared for use in a project examining the depths of SOME/IP communication or using the SOME/IP protocol for a specific reason.

Two major areas of improvement were recognized. Firstly, the service definitions need a bit of a refinement, eliminating mistakes in the source files and simplifying the description. Secondly, a more responsive and engaging GUI, or maybe outright an application, would be an excellent tool for easier testing and use of the SOME/IP protocol.

Appendix A

Bibliography

[1] Protocol numbers. https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml#protocol-numbers-1. Accessed on May 14, 2023.

[2] The world counts. https://www.theworldcounts.com/challenges/planet-earth/state-of-the-planet/world-population-clock-live. Accessed on May 13, 2023.

[3] Ieee standard for local area network mac (media access control) bridges. *ANSI/IEEE Std 802.1D, 1998 Edition*, pages 58–109, 1998.

[4] Ieee standard for local and metropolitan area networks: Media access control (mac) bridges. *IEEE Std 802.1D-2004 (Revision of IEEE Std 802.1D-1998)*, pages 137–179, 2004.

[5] AUTOSAR. SOME/IP protocol specification. https://www.autosar.org/fileadmin/standards/R22-11/FO/AUTOSAR_PRS_SOMEIPProtocol.pdf, Nov 2022.

[6] AUTOSAR. SOME/IP service discovery protocol specification. https://www.autosar.org/fileadmin/standards/R22-11/FO/AUTOSAR_PRS_SOMEIPServiceDiscoveryProtocol.pdf, Nov 2022.

[7] Vangie Beal. The 7 layers of the osi model. https://www.webopedia.com/definitions/7-layers-of-osi-model/, Apr 2022.

[8] P. Biondi. Scapy. [online] Available: https://scapy.net/.

[9] V. Cerf and R. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22(5):637–648, 1974.

[10] Michelle Cotton, Leo Vegoda, Ron Bonica, and Brian Haberman. Special-Purpose IP Address Registries. RFC 6890, April 2013.

[11] Martijn Faassen. lxml. [online] Available: https://lxml.de/.

[12] Simon Frohn and Fabian Rees. From signal to service. https://cdn.vector.com/cms/content/know-how/_technical-articles/Ethernet_AUTOSAR_Adaptive_Elektronik_Automotive_201803_PressArticle_EN.pdf, Mar 2018.

[13] Wanlin Huang and Xuesong Mao. Decision feedback equalization for enhancing plastic optical fiber transmission in automotive optical ethernet. In *Advanced Sensor Systems and Applications XII*, volume 12321, pages 227–232. SPIE, 2022.

[14] Teledyne LeCroy. Fundamentals of 100base-t1 ethernet. https://www.teledynelecroy.com/doc/100base-t1-ethernet-appnote. Accessed on May 24, 2023.

[15] Kirsten Matheus and Thomas Königseder. *Automotive ethernet*. Cambridge University Press, 2021.

[16] Nicolas Navet and Françoise Simonot-Lion. *Automotive embedded systems handbook*. CRC press, 2017.

[17] François Yergeau. UTF-8, a transformation format of ISO 10646. RFC 3629, November 2003.