CTU

CZECH TECHNICAL UNIVERSITY IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Measurement

Master's Thesis

Vehicle IP Network Analyzer

Bc. Peter Fučela
Cybernetics and Robotics

May 2024
Supervisor: Ing. Jan Sobotka, Ph.D.

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Fučela Peter**	Personal ID number:	**510635**
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:

Vehicle IP Network Analyzer

Master’s thesis title in Czech:

Analyzátor automobilové IP sítě

Guidelines:

1. Research available computer network analysis tools.
2. Implement in-vehicle IP network analysis software with the following features:
a. Network topology identification and visualization.
b. Identification of individual nodes – physical and network address, available services, and used communication protocols.
c. Identification of data flows between individual ECUs.
3. The software should primarily focus on offline analysis of measured data.
4. Explore the possibilities of network identification using active communication (ICMPv6, DoIP).

Bibliography / sources:

[1] MATHEUS, Kirsten; KÖNIGSEDER, Thomas. Automotive ethernet. Cambridge University Press, 2017.
[2] Craig Smith. 2016. The Car Hacker's Handbook: A Guide for the Penetration Tester (1st. ed.). No Starch Press, USA.
[3] 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: **30.01.2024**   Deadline for master's thesis submission: **24.05.2024**

Assignment valid until:
by the end of summer semester 2024/2025

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

Acknowledgement / Declaration

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. In addition, I would like to thank my family, friends and girlfriend for their support and patience during my studies.

I declare that this thesis is my own work and I have cited all sources I have used in the bibliography.

Prague, May 20, 2024

. . . .

Abstrakt / Abstract

Cieľom tejto práce je otestovanie možnosti rekonštrukcie topológie Ethernetovej siete vo vozidle na základe logov vytvorených jednosmerným zachytávaním paketov. Vytvorený nástroj by mal dokázať detekovať prítomnosť prepínačov, smerovačov a všetkých zariadení ktoré boli zachytené v logu. Práca rozoberá technológiu Automotive Ethernet a spôsoby, akými môžeme Ethernetové siete monitorovať. Následne predstavuje algoritmus, ktorý zostavuje topológiu jednotlivých meraných segmentov a algoritmus ktorý spája segmenty do celkovej topológie siete. Následne sú v grafe topológie zobrazené dátové toky v sieti. Pri meraní dát z viacerých segmentov paralelne sú niektoré pakety zachytené viacnásobne, čo spôsobouje zkreslené štatistiky o dátových tokoch v sieti. Preto bol navrhnutý postup ako tieto štatistiky napraviť a boli implemnetované základné štatistické funkcie po vzore Wiresharku. Záverečná kapitola sa zaoberá možnosťami vylepšenia výsledkov algoritmu pomocou aktívneho dotazovania v sieti.

Kľúčové slová: Automotive Ethernet, topológia siete, analýza dátových tokov, sieťové štatistiky.

The objective of the thesis is to test the possibility of reconstructing the topology of the in-vehicle Ethernet network based on the logs generated by unidirectional packet capture. The created tool should be able to detect the presence of switches, routers, and all end devices that have been captured in the log. The document discusses Automotive Ethernet technology and the ways in which Ethernet networks can be monitored. Then it presents the algorithm that creates the topology graph of each measured segment and the algorithm that merges the segments to the overall network topology. The data flows are further visualized in the topology graph. When packets from multiple segments are captured parallelly, some packets are captured multiple times, which causes skewed statistics about data flows in the network. Therefore, a procedure was proposed to correct these statistics, and basic statistical functions were implemented along the lines of Wireshark. The final chapter discusses how the results of the algorithm can be improved by sending the packets on the network from the test device.

Keywords: Automotive Ethernet, network topology, data flow analysis, network statistics.

Contents /

Tables / Figures

Chapter 1
Introduction

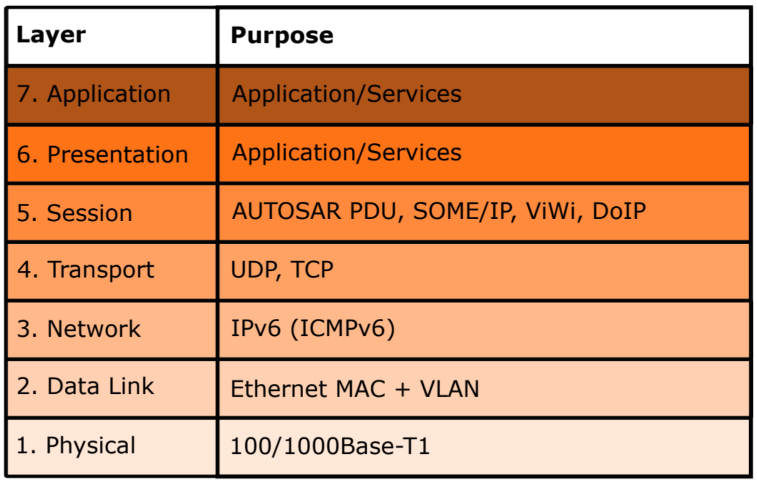
The first use of Ethernet in the automotive industry can be traced back to 2004 when Thomas Konigseder was tasked with finding a solution to speed up the software flashing process. With the CAN interface used at a time, flashing the 1 Gbyte of data would require 16 hours. After careful evaluation, Thomas chose and enabled the use of standard 100Base-TX Ethernet for flashing purposes, resulting in the first serial car with an Ethernet interface, a BMW 7-series [1].

The EMC properties of standard 100Base-TX Ethernet significantly limited its applications - the technology was usable with cost-competitive unshielded cables only when the car was stationary in the garage, meaning that the typical use-cases of Ethernet were for software flashing and On-Board Diagnostics. BMW further examined options to remove the EMC limitation with Broadcom, resulting in an optimized physical layer transceiver (PHY) that yielded even better EMC performance over unshielded cable than the FlexRay. With the EMC limitation removed, the second generation of Ethernet automotive applications started. In this generation, the Automotive Ethernet is commonly used for infotainment and vehicle sensors such as cameras, LiDARs, and other sensors used for ADAS. In the future, the third generation of Automotive Ethernet can be expected. While in the previous generations, the Ethernet was responsible only for specific application domains, in this generation, it will become the backbone of the in-vehicle network [2].

It is evident that proper functioning of the Ethernet network in a car is critical for ensuring vehicle safety and security, and its importance will only increase in the near future. Our university, together with TUV Sud and Skoda Auto, is working on a solution for testing and analyzing vehicle networks. As part of this project, this thesis investigates the possibility of reconstructing the network topology based on packet captures that are sniffed on Ethernet interfaces. The motivation for exploring this topic is that external companies that want to test or certify vehicles often do not have access to the documentation of the devices, network topology, and protocols used in the vehicle. The network analyzer tool should provide them with this information.

Chapter 2
Ethernet network analysis background

Since its invention in the early 1970s, Ethernet has become the most common form of wired LAN in the computer world. This is one of the reasons for the use of technology in the automotive environment. Broadcom Corporation, one of the leading semiconductor design firms in the field of communication and networking, developed OABR (OPEN Alliance BroadR-Reach) Ethernet technology in 2011 [2]. The OPEN Alliance (One-Pair Ether-Net) Inc. is a non-profit group formed in 2011 by mostly automotive industry and technology providers collaborating to promote wide-scale adoption of Ethernet-based networks as the standard in networking applications [3].



Layer	Purpose
7. Application	Application/Services
6. Presentation	Application/Services
5. Session	AUTOSAR PDU, SOME/IP, ViWi, DoIP
4. Transport	UDP, TCP
3. Network	IPv6 (ICMPv6)
2. Data Link	Ethernet MAC + VLAN
1. Physical	100/1000Base-T1

Figure 2.1. ISO/OSI model of Automotive Ethernet.

BroadR-Reach technology allows point-to-point Ethernet communication over a single unshielded twisted pair (UTP) cable. As wiring is one of the heaviest and most expensive components of a modern car, using as few cables as possible directly impacts vehicle price and fuel economy.

OABR Ethernet was standardized by OPEN Alliance SIG in 2015 into IEEE standard IEEE 802.3bw also known as 100BASE-T1. After a year in 2016 the standard IEEE 802.3bp was published for 1Gb/s Physical Layer also known as 1000BASE-T1.

2.1 Physical layer

In order to meet the demands of the automobile sector, special physical layers were created to enable Ethernet in vehicles. One of the factors taken into account was the desire to employ unshielded twisted pair (UTP) cables in order to guarantee a high level of electromagnetic interference immunity, while also saving weight and money, as wiring is one of the heaviest and most expensive components in a modern vehicle.

The physical layer can be changed as needed without affecting the layers above it. This greatly improves the flexibility of the network. Higher data rates can be easily implemented by using a different physical layer. Currently 3 different Physical Layers are used in the automotive industry, described in sections below.

2.1.1 100 BASE TX

- Standard physical layer used in the PC field
- In vehicles it is used for the diagnostic interface
- As a physical medium it uses 4-wire shielded cable
- Data transfer rate is 100 Mbps
- Maximum length of a cable is 100 meters

2.1.2 100 BASE T1

- Special physical layer for use in the automotive sector
- In vehicles it is used mostly for the networking of ADAS ECU's
- As a physical medium it uses unshielded 2-wire cable
- Data transfer rate is 100 Mbps
- Maximum length of a cable is 15 meters

2.1.3 1000 BASE T1

- Special physical layer for use in automotive sector
- In vehicles it is used mostly for networking of ADAS ECU's and infotainment systems
- As a physical medium it uses 2-wire unshielded cable
- Data transfer rate is 1000 Mbps
- Maximum length of a cable is 15 meters

One more important property of Physical Layers mentioned above is that all of them are full duplex, which is not the case with some older automotive networks such as CAN, Flexray or MOST. This means that while each 100 BASE T1 link is rated at 100Mbps, the total aggregate throughput between nodes is 200Mbps [4].

Another difference from traditional automotive networks is that, at the basic Physical Layer level, the Ethernet network is constructed of point-to-point lines, meaning that each UTP cable supports only two nodes, one at each end. Considering a typical network containing more than two devices, a switch is used to interconnect the devices.

To illustrate this, consider a simple 100 BASE T1 network with 4 nodes shown in Figure 2.2. There is a switch with four ports connected to the head unit, the display node, the console node, and the speaker node. Each UTP cable from the switch to the four end nodes is physically distinct from the other; it could be said that there are four mini-networks here, called network segments.

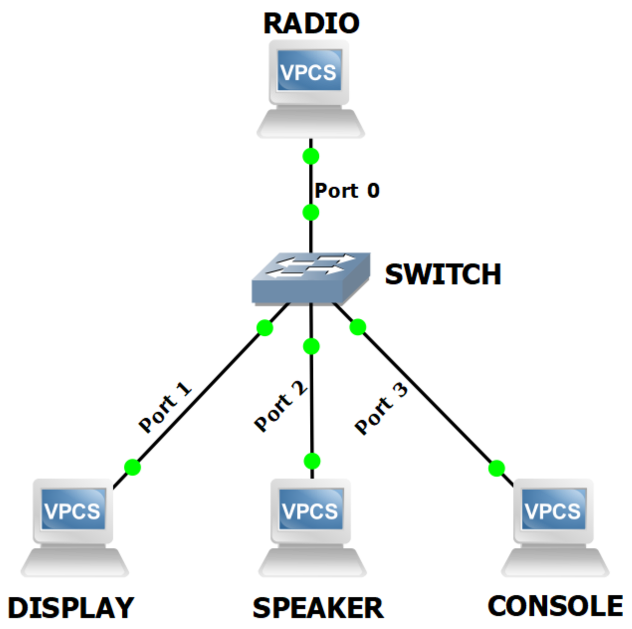


Figure 2.2. Example of Automotive Ethernet network with 4 nodes and 1 switch.

The advantage of a full-duplex, packet-switched network as Ethernet is its ability to support multiple data exchanges between nodes simultaneously [5]. For example, consider that the display needs to communicate with the console, while the head unit needs to talk with the speaker. Because all links are independent and the switch transmits received messages only to the specified recipients, these two communications can occur concurrently without waiting. When each device transmits at the full nominal rate of 100 Mb/s, the total aggregate network bandwidth is 400 Mb/s.

2.2 Data Link Layer

The data link layer is left unchanged. It provides the basic functions for bus traffic control and bus access control and defines a unified structure of communication frames, including how nodes are addressed. All these basic functions are implemented in the Ethernet controller, which is usually integrated in the microprocessor.

For data transmission and node communication, unique identification addresses are essential. Under the IEEE 802 scheme, the MAC address consists of six bytes. The initial three bytes (OUI field) are designated to hardware manufacturers, while the subsequent three bytes are assigned by the manufacturer to their respective products.

Devices often require data to be transmitted to multiple recipients simultaneously. To accommodate this, the IEEE 802 MAC scheme uses a specific flag that distinguishes transmissions directed at a group of recipient MAC addresses from those aimed at a single target. This flag is located in the first (least significant) bit of the OUI field of the MAC address, known as the I/G (individual/group) flag. When this bit is set to zero, indicating an individual device, the message is sent as a unicast. Conversely, when set to one, it signifies a group address (multicast). Since the I/G bit is exclusively set to one for group addressing, it is only applicable as a target address.

Another specific MAC address is the broadcast address, which is essentially a MAC address with all bits set to one: `FF-FF-FF-FF-FF-FF`. Within a LAN, any frame that has this MAC address in its destination address is intended for any device that might be listening. Broadcasts serve to address specific scenarios in which the sender lacks knowledge of the address of the device with which it needs to communicate.

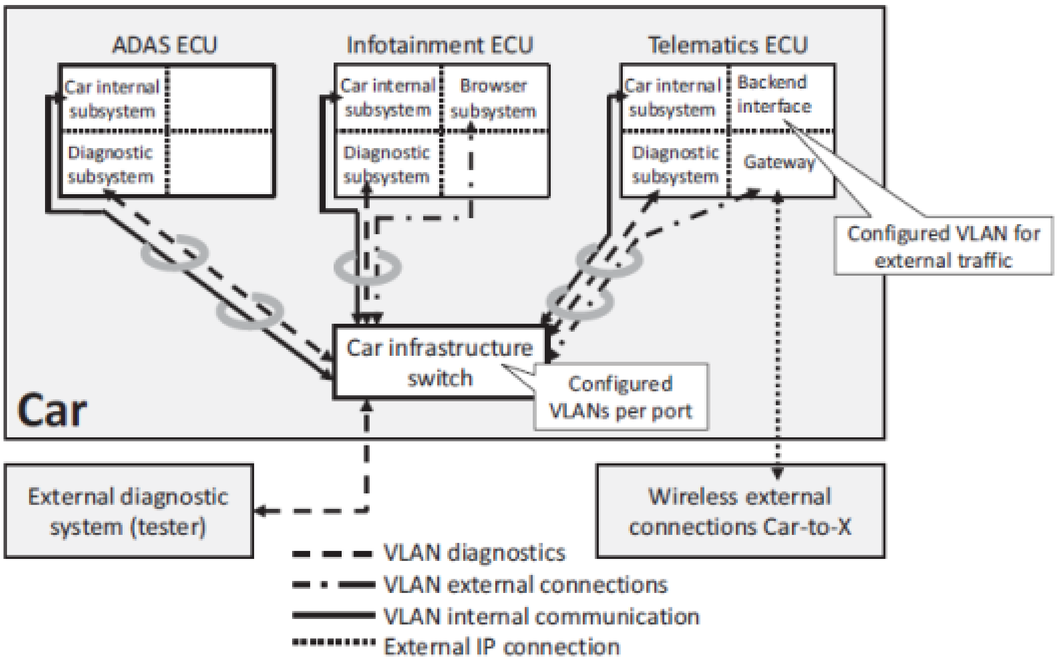


Figure 2.3. VLANs example [1].

In automotive applications, the traditional addressing scheme can be enhanced by employing Virtual Local Area Networks (VLANs). VLAN addresses facilitate the creation of virtual networks on top of a physical one, allowing the segmentation of communications. This segmentation defines distinct domains for various applications and use cases. An Electronic Control Unit (ECU) can belong to multiple application domains and VLANs as shown in Figure 2.3. In addition, VLANs serve as a foundation for security measures that incorporate firewall functionality. For example, this setup ensures that specific browser applications cannot access internal car data, even if transmitted over the same network wire [1].

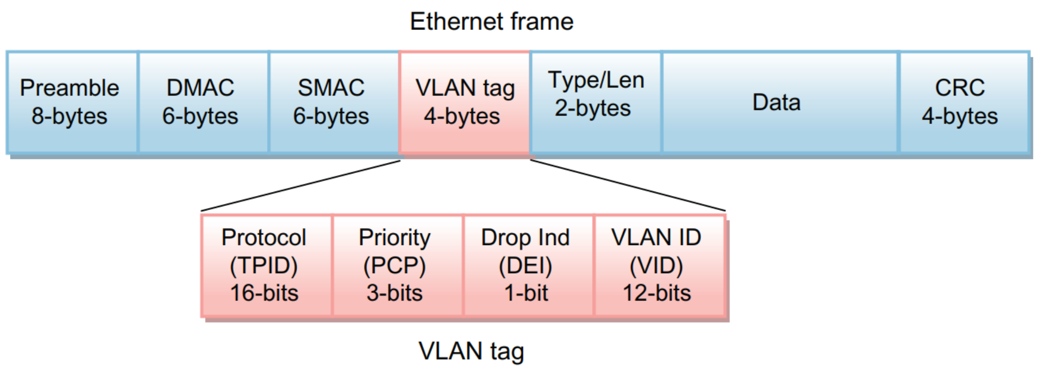


Figure 2.4. VLAN tag. [6]

It is worth noting that VLAN also allows to assign distinct priorities to each message routed to Ethernet switches, enhancing real-time communication capabilities. This is done using a process called `Q-tagging`. To tag the frame, 4 bytes are inserted between the Source MAC address and the Length/Type field in the 802.3 header. These 4 bytes identify the frame as being VLAN tagged and provide the necessary tag information. The first two bytes are called tag protocol identifier (TPID), which highlights that the frame is tagged by using a special Ethertype, which has a value of 0x8100. The other two bytes are called tag control information (TCI), which is further divided into 3 subfields:

- Priority Code Point (PCP): Indicates the priority of the frame. The size of the field is 3 bits, meaning that there are eight priority levels, where a higher value means a higher priority.
- Drop Eligible Indicator (DEI): A flag that indicates whether the frame is suitable for being dropped in the event of network congestion (is flag is set to 1).
- VLAN Identifier (VID): This subfield is 12 bits long, meaning that there are 4096 possible VLAN IDs. Of these, three are reserved. VID of 0 indicates that the frame is not assigned to any VLAN, in this case the tag serves only to specify the priority. VID of 1 is the default value, and the VID of 4095 is reserved.

Figure 2.3 illustrates an example of a vehicle network with VLANs, where the internal car communication, diagnostics, and external connections run on separate VLANs.

2.3 TCP/IP

TCP/IP combines layers 3 and 4 within the Ethernet OSI model. Just as in regular LAN setups, the TCP/IP suite operates over the Ethernet physical layer to support various automotive applications. In the following sections, the individual protocols that make up the TCP/IP suite are explained.

2.3.1 Internet protocol

The Internet Protocol (IP) handles the routing between different networks. While Ethernet frames within a network can be sent directly via a switch, packets destined for hosts in remote networks must pass through a router. To facilitate this, each network node is assigned a unique IP address, which operates independently of the physical address (MAC address) and thus acts as a logical address. When a packet is sent to a host not lying in the same network, the router can look up the network this node lies in and can thus route the packet to the correct network. For automotive use, routing is commonly used to send packets between different VLANs.

Although the number of active nodes in an in-vehicle network can change, the network is considered a closed system because the maximum number of nodes is predetermined. Due to the frequent restarts a car undergoes daily, there is a time constraint that requires swift initialization of all nodes. Therefore, static IP configurations are usually recommended and used [2].

IP addressing can work in four different modes, according to the intended recipients, as shown in Figure 2.5. The possible modes are:

- Unicast - the connection between two devices.
- Anycast - the message is intended only for any recipient in a specific group.
- Broadcast - the message is sent to all recipients inside a network
- Multicast - the message is sent to an arbitrary number of recipients

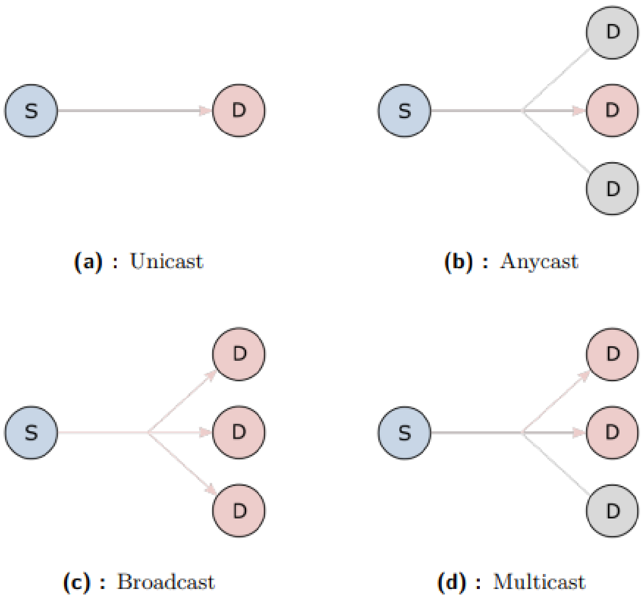


Figure 2.5. IP addressing modes. [7]

2.3.2 IPv4

Figure 2.6 displays the IPv4 header format.

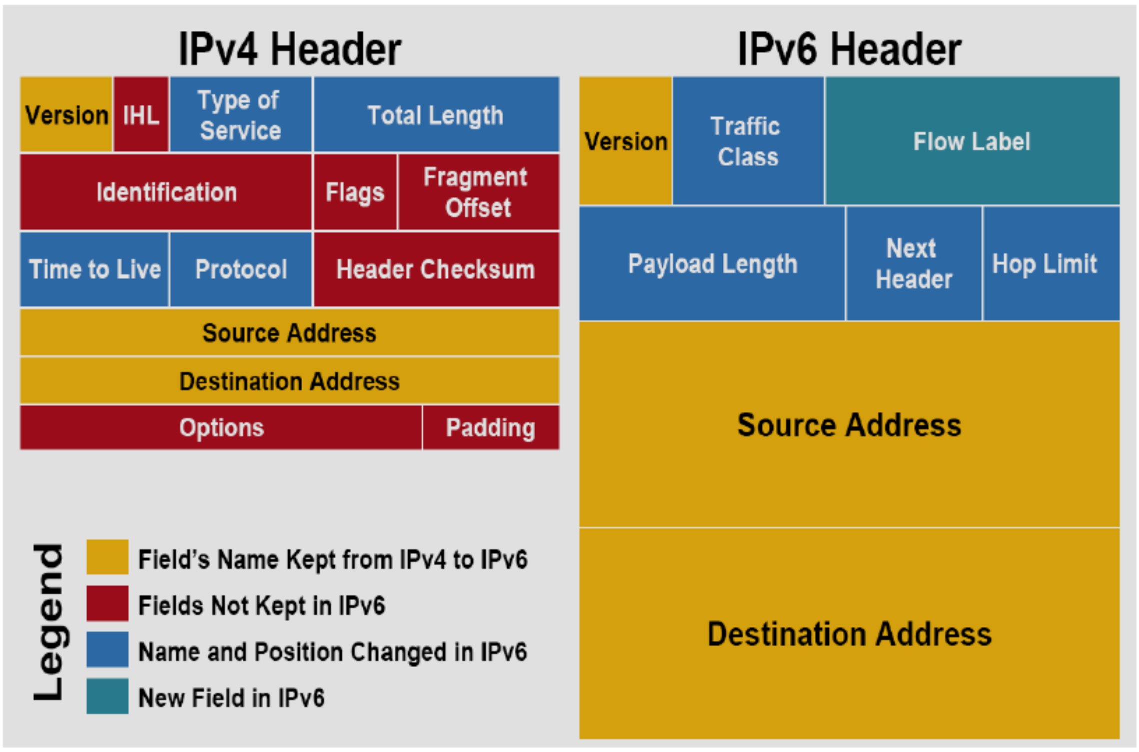


Figure 2.6. Differences in headers between IPv4 and IPv6. [8]

The header fields are explained below:

- Version: specifies the version of the IP protocol (IPv4).

- IP Header Length (IHL): specifies the length of the header in multiples of 4 bytes.
- Type of Service (TOS): specifies the quality of service, for example, delay or precedence.
- Identification, Flags, Fragment Offset: these fields are used to facilitate IP fragmentation. This process is used to break down large IP packets into smaller ones when endpoints or routers cannot handle large IP packets. However, modern routers typically no longer support IP fragmentation. Instead, they discard such packets and send a control message back to the source Electronic Control Unit (ECU) to notify the transmitter that the selected packet size is not supported.
- Time to Live (TTL): When an IP packet traverses a router, the TTL field is reduced by 1. Upon reaching 0, the router discards the packet and sends an ICMP time exceeded message back to the sender.
- Protocol: This 8-bit field indicates the protocol encapsulated within the IP packet. For most common protocols, TCP is assigned a value of 6, while UDP is assigned a value of 17.
- Header Checksum: contains checksum that allows receiver to check whether there are errors in the header
- Source Address: 32 bit address of the sender
- Destination Address: 32 bit address of the receiver
- IP options: optional field for more specific uses of the protocol
- Padding: used to ensure that the header size is multiple of 4 bytes (due to IHL)

2.3.3 IPv6

The changes in the IPv6 header structure, illustrated in Figure 2.6, begin with the elimination of the IHL field and the replacement of the TOS with the Traffic class. Subsequently, the total length is replaced by the payload length, which now excludes length of the header, and is positioned after the Flow label field. The Flow label provides an option to label a sequence of IPv6 packets. The Next header field denotes the type of the encapsulated protocol. The final modification to the IP header, aside from the address sizes, it the Hop limit field, which replaces the TTL field while serving practically the same function.

2.3.4 TCP

In the transport layer of the Internet, there are two primary protocols: one is connectionless, and the other is connection-oriented. These protocols complement each other. TCP, the connection-oriented protocol, handles many tasks such as establishing connections, ensuring reliability through retransmissions, and managing flow and congestion control, serving the applications that utilize it.

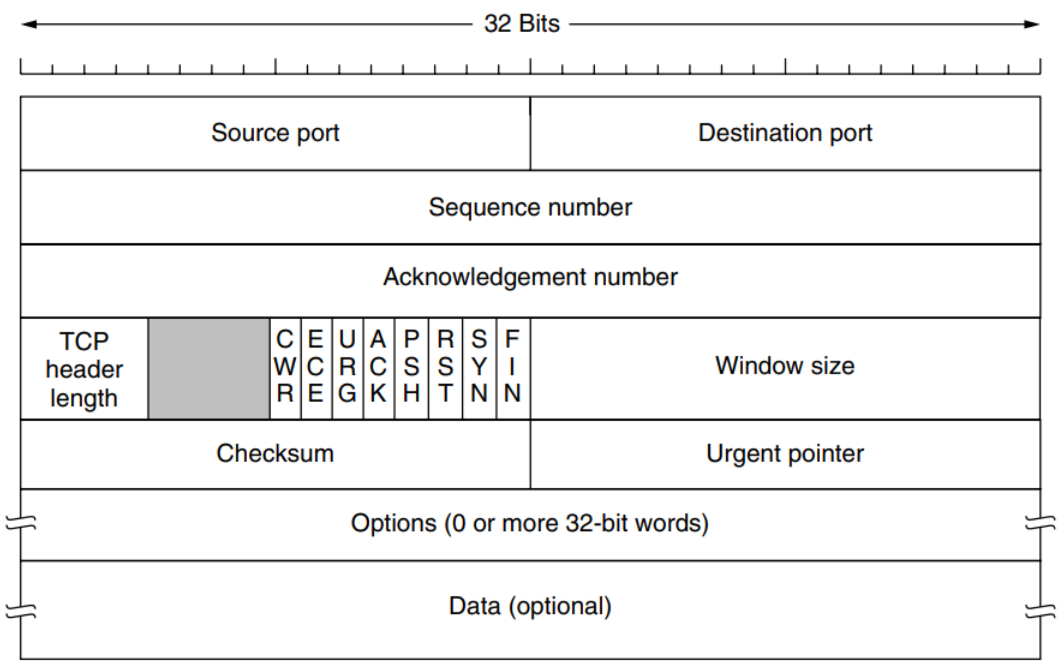


Figure 2.7. The TCP header. [9]

A TCP connection is started through a three-way handshake where, firstly, ECU A sends a request for connection establishment to ECU B, specifying the Sequence Number. The ECU B sends the acceptance of the connection, also specifying the sequence number. Finally, ECU A sends the final acceptance acknowledgment. Once established, data can flow between the two nodes via a point-to-point connection; in other words, TCP does not support multicast or broadcast. TCP further confirms all successfully received packets, using a checksum to identify transmission errors, and retransmits incorrectly transmitted segments. The receiving device assembles the transmitted segments using sequence numbers [9]. This makes the structure of a TCP header more complex than its UDP counterpart, as seen in Figure 2.7.

2.3.5 UDP

The purpose of the UDP (User Datagram Protocol) is to provide a lightweight communication protocol that allows the exchange of data between devices without the overhead of establishing a connection, with error checking performed only at the nodes. This protocol operates without establishing connections, allowing out-of-sequence deliveries and lacking segment numbering for loss checks. However, having connectionless transmission results in faster transmission compared to TCP. Additionally, UDP enables segment transmission in multicast or broadcast mode. Figure 3.21 displays a header for the UDP segment.

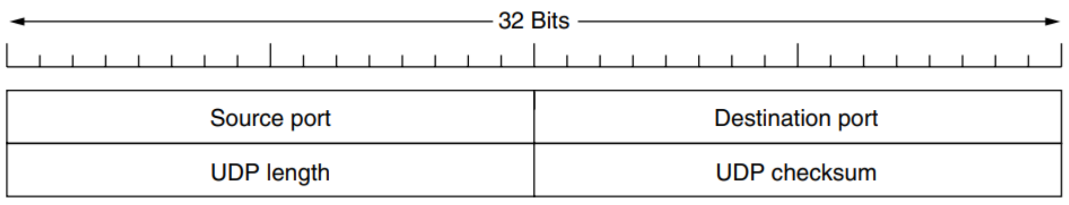


Figure 2.8. The UDP header. [9]

In addition to the data payload, UDP includes information about the source and destination ports, each represented by 16-bit integers ranging from 0 to 65,535. Many of these ports are reserved or forbidden. The length field in UDP summarizes the size of both the UDP header and the carried payload. For error detection, a checksum field is appended to the header. To generate the checksum, the sender calculates the sum of all 16-bit segments in the packet, then inverts all the bits of the sum, and inserts the resulting 16 bits into the checksum field. The receiver computes the checksum in a similar manner, adding its own checksum to the sender's checksum. If the result contains any zeros, an error is detected and the receiver discards the packet.

2.3.6 Topology

Because an Automotive Ethernet network with more than two nodes requires a switch to interconnect its end devices, it is naturally based on a star topology. Adding more connections allows the star topology to be easily expanded, limited only by the number of ports on the switch. Multiple connected switches can extend the topology even further into a tree topology [2].

The example shown in Figure 2.2 is a fundamentally different way of networking than that implemented by CAN, FlexRay, LIN, or MOST. These older networking technologies use a shared medium, meaning that all devices can access the cabling through which they are connected. Shared cabling prevents more than one device from communicating on the network, but it also means that if any frame exists on the network, all connected nodes can see it. However, in a switched Ethernet network, the sender usually sends the frame to a switch, which will then retransmit it only on the port connected to the intended recipient; the remaining ports will remain idle.

As an example, let us return to the example from Figure 2.2. If the console transmits a frame to the display, the frame will first be sent to port 3 of the switch. After a while, the switch will copy the frame to its port 1, from which it will travel to the display. This has two consequences; first, for most of the transmission, parts of two frames will exist on the network: one traveling from the console to the switch and another from the switch to the display. Second, the frame will never exist on the network segments leading to devices that are neither source nor destination; ports 0 and 2 will remain idle, as will the cables attached to them.

There are more consequences when it comes to the selection of tools in the automotive network. CAN tools cannot be used to monitor or simulate message traffic, as there is no single place on the network to see all the traffic. Connecting a tool to a link on an Ethernet network only shows the frames on that link, not everything on the network, as is the case with older technologies.

2.4 Network Traffic Monitoring

There are many different approaches for monitoring Ethernet networks. The selection of the methods will depend on available hardware and requirements of the user. Three common methods are explained in the following sections.

2.4.1 Monitoring network using switch port mirroring

In conventional Ethernet, switches often deal with the task of traffic monitoring using a technique called port mirroring. The principle of the method is that a copy of each frame that passes through a switch is mirrored to a debug port, or to a regular port configured for this purpose. A computer or other device can be connected to this port and use standard software to sniff packets, such as Wireshark, to monitor the switch traffic. A regular computer lacks support for Automotive Ethernet Physical Layers like BroadR-Reach, so it is necessary to convert packets at Layer 1 to enable connection to a conventional Fast or Gigabit Ethernet port on the monitoring device[2, 10].

This method seems to be promising; however, for monitoring Automotive Ethernet, there are several drawbacks:

- All faulty frames on the network will be dropped by the switch, and they won't be forwarded to the debug port. It is impossible to identify defective frames on the network using this method.
- As explained earlier, the total aggregate throughput of the network is much higher than the throughput of a single link. Unless the debug port is magnitudes faster than a regular port, overflow may occur, where the port buffer fills up, and additional frames are discarded. This means that the user will never see these frames.
- The existence of a fast enough debug port makes the system more expensive. It is necessary to have a physical connector accessible to the monitoring device, and all associated Physical Layer chips and connections on a board must be provided. Car manufacturers usually don't provide additional hardware to a production vehicle just for debugging.

2.4.2 Single active test access point

Another option is to use an active test access point (TAP), a device developed to test Ethernet networks. An active TAP is inserted into a link of an Ethernet network to provide access to the network for testing. It must be understood that the TAP is not simply connected to the physical wires of the network in the same way as when the CAN network is measured using the Y-cable. Instead, it is inserted, which means that the wires connecting the TAP to the switch (Figure 2.9) are physically separated from the wires connecting the TAP to the Monitoring node. The TAP will transfer all Ethernet frames from Port 1 to Port 1' and all frames from Port 1' to Port 1. The TAP will also copy these frames to a port connected to a monitoring node so that a PC can monitor them using a test tool. The PC can also transmit Ethernet frames through the TAP [11].

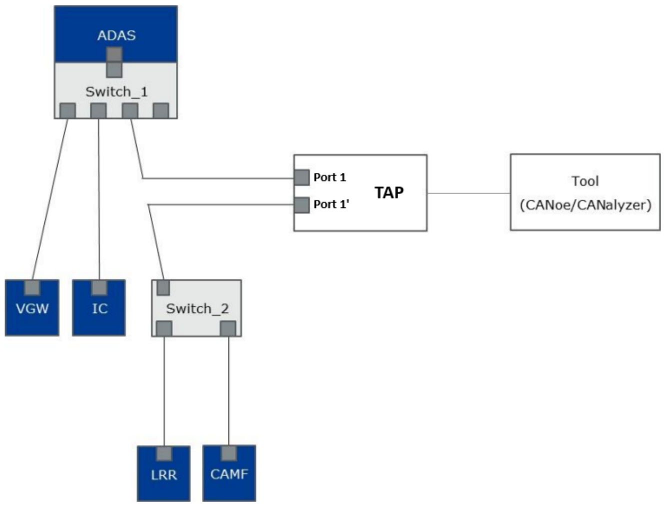


Figure 2.9. Example of single active TAP connection. [11]

The benefit of this method is that the TAP can capture a corrupted Ethernet frame and encapsulate it into a new frame. This frame can be sent to the computer, which will unwrap the corrupted frame and allow us to analyze it. With this method, packets can also be sent on the network with the same MAC address as the node to which the TAP is connected, without introducing a new MAC address to the network.

The main drawback is that the user must be able to insert the device into the network, and the TAP introduces a delay in the network by copying the frames from one port to another.

2.4.3 Multi active test access point

The principle of multi active TAP is same as with single active TAP, resulting in the same advantages and drawbacks. The added benefit is that it allows monitoring and synchronizing frames from multiple ports of an Automotive Ethernet switch. Modern devices for monitoring Automotive Ethernet supports monitoring up to 12 switch ports.

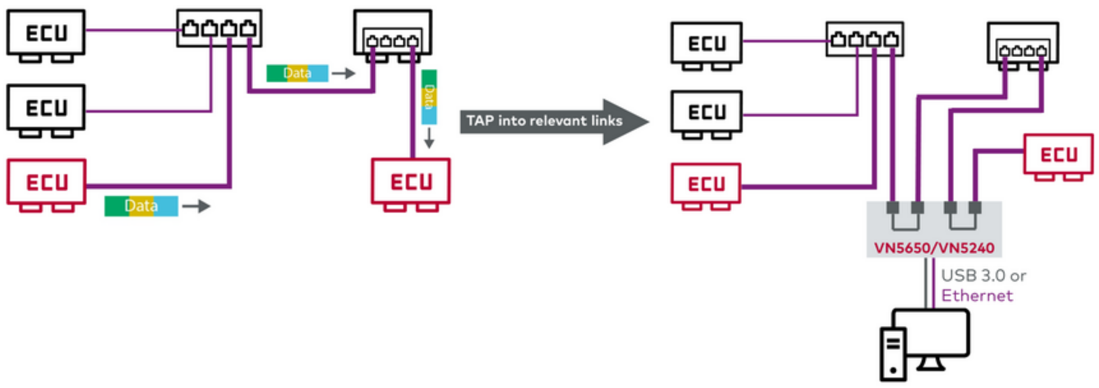


Figure 2.10. Example of multi active TAP connection.[11]

2.5 Network Traffic Analysis

This section presents data sources for network traffic analysis and present example of some popular tools for network analysis.

2.5.1 Data Sources

Two primary sources of data for analyzing network traffic include packet captures (PCAPs) and network flows (NetFlows). NetFlows represents connections between communicating nodes. In contrast to PCAPs, NetFlows contains limited attributes, typically specifying the source and destination of the flow, its volume, and selected details about communication protocols. In contrast, PCAPs are used for in-depth examinations of network performance issues or forensic analyses of identified cybersecurity incidents. As implied by its name, PCAPs represent the unprocessed packet data extracted from the network or its segments.

2.5.2 Common Network Analysis Tools

Network testers and security analysts use command-line utilities or highly flexible tools that allows user to capture and inspect the network traffic. Common examples of command-line utilities used in traffic analysis is `ping` or `tracert` (`traceroute` for UNIX-like systems):

- Ping command uses the Internet Control Message Protocol (ICMP) to verify connectivity to another TCP/IP device at the IP level. It sends echo Request messages to the target IP address, which responds by sending back an echo Reply message. The received message is displayed, together with round-trip times. Ping is an essential tool for troubleshooting connectivity, reachability, and name resolution [12].
- Traceroute command also uses ICMP echo Request messages, but this time it modifies the TTL field (Section 2.3.2) of the IP packet. First, it sends a packet with TTL of 1 and increments it by one on every subsequent transmission, until the target device responds back with an echo Reply message. Each router along the path is required to decrease the TTL in an IP packet by 1 before forwarding it. When the TTL reaches 0, router sends Time exceeded message back to the sender, letting the program know that there is a router in the path [13].

Typical examples of more advanced open-source tools for network traffic analysis are Wireshark or Network Miner:

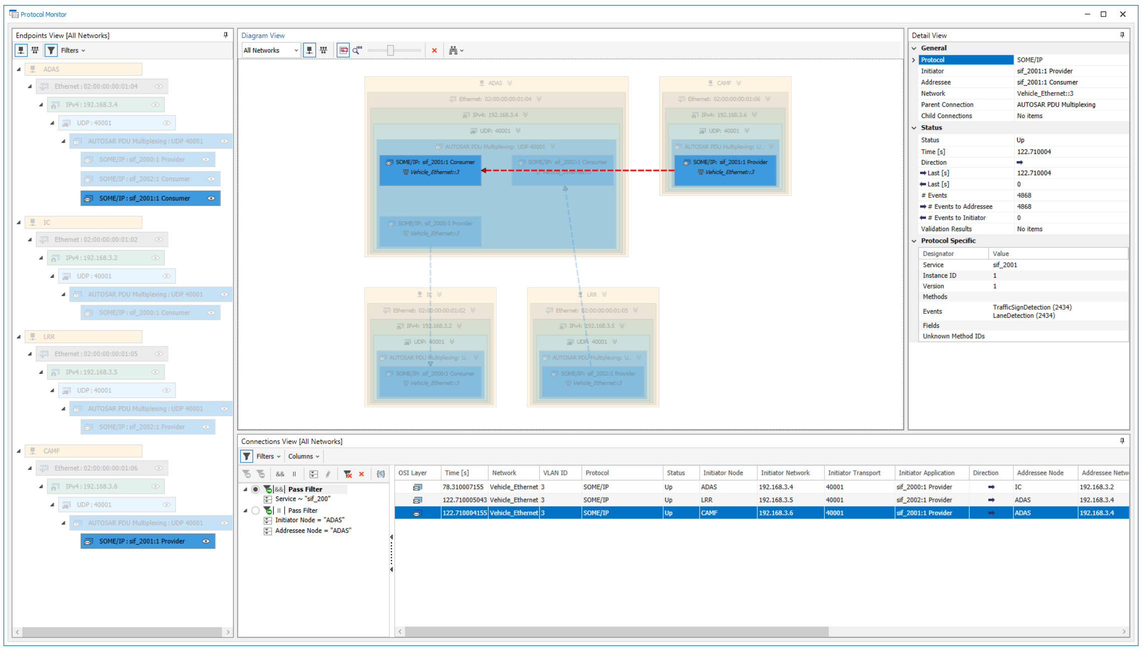
- Wireshark [14] is the most popular network packet analyzer. It allows the user to capture packets and analyze them both online and offline. For analysis it provides functions for advanced filtering and inspection of the protocols.
- NetworkMiner [15] is a network forensics application specifically designed to retrieve artifacts from network traffic captured in PCAP files, such as files, images, emails, and passwords. Similarly to Wireshark, NetworkMiner can also capture live network traffic by monitoring a network interface.

Both tools offer network statistics and detailed information for each captured packet. However, they display the data primarily in tables with limited visualization features, such as highlighting rows or basic static line charts.

In the automotive domain, one of the most significant providers of network analysis tools is Vector GmbH, providing both hardware (test access points) and software. Their software CANoe provides tools for analysis, diagnostics, simulation, and testing of the automotive networks. Specifically for analysis of the Ethernet traffic, CANoe Ethernet [16] implements two tools, Trace Window and Protocol Monitor:

- Trace Window provides very similar functionality to the packet analyzers mentioned above. It allows the user to capture packets in real-time, display them during measurement, filter, and search for specific packets, decode packet contents, and inspect packet details.
- Protocol Monitor, shown in Figure 2.11 provides a real-time visual representation of Ethernet traffic. It shows the endpoints on all OSI layers and

the communication relationships between them. It also provides filtering functions to view only certain types of network communications.



Detail View

General

Protocol	SOME/IP
Initiator	sif_2001:1 Provider
Addressee	sif_2001:1 Consumer
Network	Vehicle_Ethernet::3
Parent Connection	AUTOSAR PDU Multiplexing
Child Connections	No items

Status

Status	Up
Time [s]	122.710004
Direction	➡
➡ Last [s]	122.710004
⬅ Last [s]	0
# Events	4868
➡ # Events to Addressee	4868
⬅ # Events to Initiator	0
Validation Results	No items

Protocol Specific

Designator	Value
Service	sif_2001
Instance ID	1
Version	1
Methods	
Events	TrafficSignDetection (2434) LaneDetection (2434)
Fields	
Unknown Method IDs	

Pass Filter
Service ~ "sif_200"
Pass Filter
Initiator Node = "ADAS"
Addressee Node = "ADAS"

OSI Layer	Time [s]	Network	VLAN ID	Protocol	Status	Initiator Node	Initiator Network	Initiator Transport	Initiator Application	Direction	Addressee Node	Addressee Netw
	78.310007155	Vehicle_Ethernet 3		SOME/IP	Up	ADAS	192.168.3.4	40001	sif_2000:1 Provider	➡	IC	192.168.3.2
	122.710005043	Vehicle_Ethernet 3		SOME/IP	Up	LRR	192.168.3.5	40001	sif_2002:1 Provider	➡	ADAS	192.168.3.4
	122.710004155	Vehicle_Ethernet 3		SOME/IP	Up	CAMF	192.168.3.6	40001	sif_2001:1 Provider	➡	ADAS	192.168.3.4

Figure 2.11. CANoe Protocol Monitor [16]

2.6 Tools for visualization of PCAPs

As the volume of network data exchanges increases, the importance of researching and creating network visualizations also grows. Visualizations are increasingly applied across various domains of network security. There are many different tools and approaches for visualization; the overview below shows mostly the graph-based tools.

NetCapVis [17], shown in Figure 2.12, offers both overview and analytics functions by enabling filtering based on transport protocol, IP addresses, and port numbers for both incoming and outgoing traffic. Users can also export filter configurations for use in Wireshark.

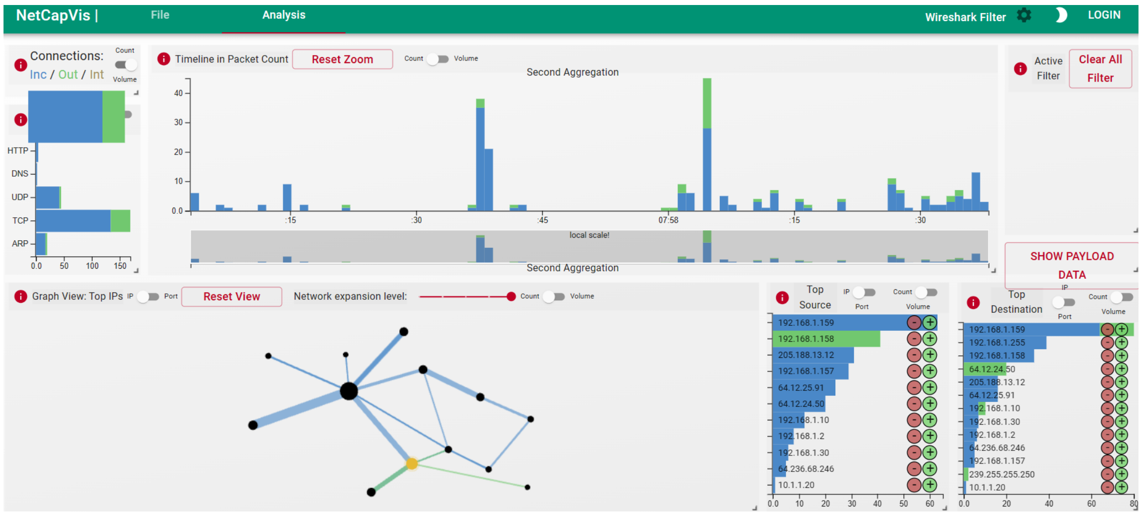


Figure 2.12. NetCapVis user interface.

A-Packets [18], shown in Figure 2.13, offers various views on the PCAP files. These views are independent of each other and mostly text-based. The network graph does not provide many filtering functions and it may be challenging to interpret for larger datasets. Under the free plan, the maximum allowable file upload size is 25 MB, and any uploaded PCAP file becomes publicly accessible.

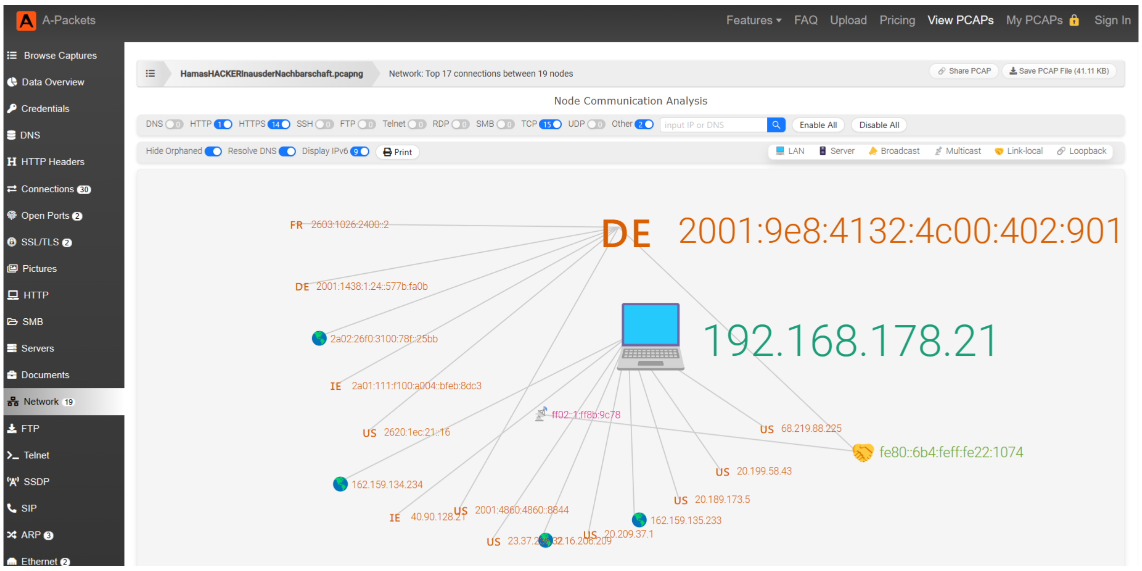
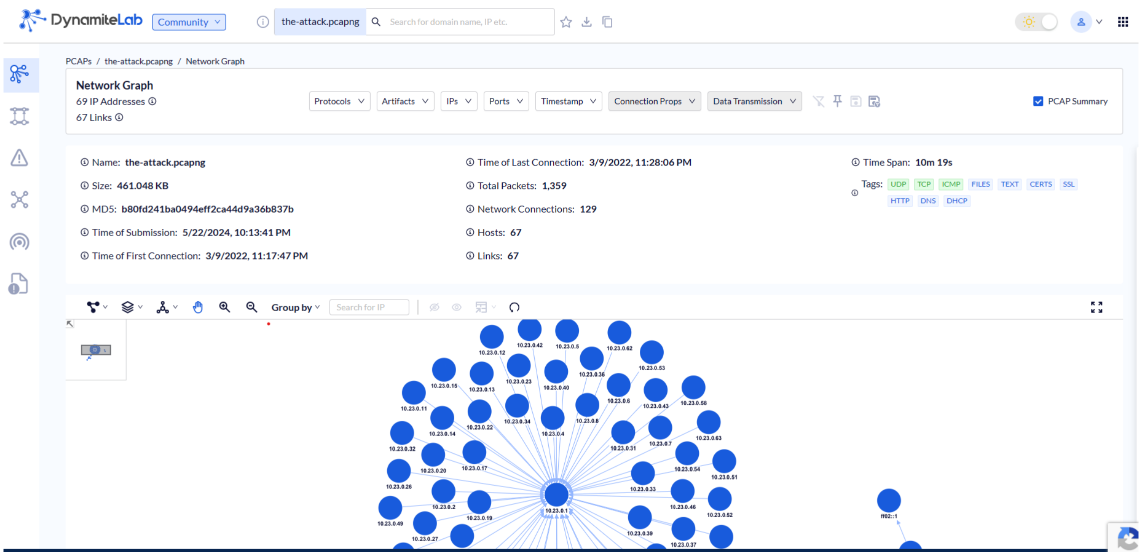


Figure 2.13. A-Packet user interface.

DynamiteLab [19], shown in Figure 2.14, which allows the user to analyze files up to 75MB, although it keeps PCAP files public as well. From network graph perspective, compared to A-packets it gives user more options for filtering - namely by protocol, IP address, ports, and timestamps.

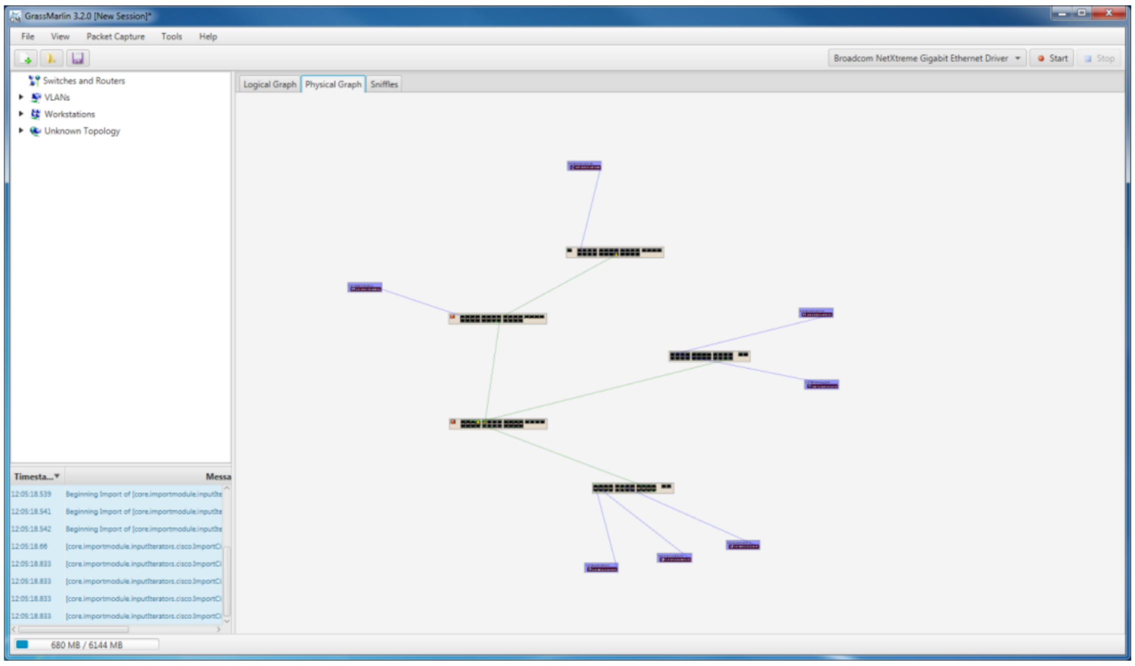


Name: the-attack.pcapng
Size: 461.048 KB
MD5: b80fd241ba0494eff2ca44d9a36b837b
Time of Submission: 5/22/2024, 10:13:41 PM
Time of First Connection: 3/9/2022, 11:17:47 PM
Time of Last Connection: 3/9/2022, 11:28:06 PM
Total Packets: 1,359
Network Connections: 129
Hosts: 67
Links: 67
Time Span: 10m 19s
Tags: UDP TCP ICMP FILES TEXT CERTS SSL HTTP DNS DHCP

Figure 2.14. DynamiteLab user interface.

GrassMarlin [20], developed by the National Security Agency, is a mapping tool tailored for industrial control systems and SCADA networks. It excels in handling large volumes of traffic data, helping to understand how data enter and move within critical environments by visualizing the network in a communication graph.

GrassMarlin, compared to previous tools, also provides a physical graph (Figure 2.15) that depicts the physical network infrastructure, including managed switches and routers, the connections between them and the physical workstations. GrassMarlin gets this data from the configuration data for managed devices [20].



```
12:05:18.539  Beginning Import of [core.importmodule.inputthe
12:05:18.541  Beginning Import of [core.importmodule.inputthe
12:05:18.542  Beginning Import of [core.importmodule.inputthe
12:05:18.66   [core.importmodule.inputiterators.cisco.ImportCi
12:05:18.833  [core.importmodule.inputiterators.cisco.ImportCi
12:05:18.833  [core.importmodule.inputiterators.cisco.ImportCi
12:05:18.833  [core.importmodule.inputiterators.cisco.ImportCi
12:05:18.833  [core.importmodule.inputiterators.cisco.ImportCi
```

Figure 2.15. GrassMarlin physical graph. [20]

2.7 Application Background

The idea of the thesis is to explore whether it is possible to use packet capture files acquired using a single-TAP or multi-TAP device to reconstruct the topology (similar to physical graph used by GrassMarlin shown in Figure 2.15) of the in-vehicle network. The TAP devices allow one to separate and tag the traffic by direction, which helps us to identify on which side of the physical interface the devices are located.

The core principle of device identification is simple, on every LAN (or VLAN) each ECU should have a single MAC address and single IP address (only network with static IP addresses is considered, as the in-vehicle network is fixed). When TAP is inserted into a wire, based on the number of different IP and MAC addresses on each side of the wire, the application should be able to detect end devices, switches working on MAC level, as they do not manipulate packets but allow multiple distinct MAC adresses and IP addresses to be on one side of the wire, and also the routers, which changes the source MAC address of packets incoming from other LANs, resulting in packets with distinct IP addresses but the same MAC address on one side of the segment.

As it is not possible to reconstruct the entire network topology only from a measurement at a single wire on the network, the algorithm that combines topology graphs constructed from multiple measurements must be developed.

The topology graph can be used for validating the network architecture with the documentation or for visualization of data-flows on the network.

Chapter 3
Input Data Analysis

This chapter explains how to get the data for analysis and what are the assumptions.

3.1 Input data

While writing the thesis, a vehicle that could be measured was not accessible and only one trace measured on a car was available. Therefore, a GNS3 network simulator was used to simulate different network topologies and validate our algorithms.

GNS3 is an open-source graphical network simulation tool that allows emulation of complex computer networks. GNS3 uses several emulators, such as Dynamips for emulating Cisco IOS, VirtualBox for emulating Windows or Linux workstations, or Pemu for emulating Cisco PIX firewall. Because emulation is used instead of simulation, the behavior of the devices is very similar to that of the real ones. One disadvantage of GNS3 is that the Dinamips emulator does not support emulation of L2 switches due to the high computational demands of emulating ASICs used in switches. A built-in switch was used to simulate switches, which is not an emulation of an actual device, but it allows some basic settings, such as setting up VLAN. For routers, image of the Cisco 7200 router was used, and for simulating end devices, a built-in Virtual PC Simulator (VPC) was used. Virtual PC Simulator is a program written by Paul Meng that allows us to simulate a lightweight PC supporting DHCP and ping. It consumes only 2MB of RAM per instance and requires no additional image [21].

GNS3 works on a client-server principle, that allows us to run server side on a different device than a client part. This is beneficial when a client computer does not have enough hardware to run multiple images and allows multiple people to collaborate on a project.

After building the network and configuring all devices, packet capture can be started by right-clicking on one of the links and selecting `Start capture`. The button will open Wireshark and start capturing packets on the interface. The traffic was simulated by opening the terminal of one of the virtual PCs and using the ping command that supports ICMP, TCP, and UDP protocol.

There is a limitation: by default, it is impossible to tag packets by their direction, which is a necessary condition for our purposes. This can be overcome by filtering the packets in Wireshark according to the network's known topology. For example, consider that traffic is captured at link *eth1.5* in the network shown in figure 4.1. To get traffic only in one direction, one can filter all frames with Node 1.1 and Node 1.2 as sources (by IP address). To get the other direction, filter them as a destination. Next, save the filtered packets in two PCAP files by their direction, and in the end, merge them using Tracewrangler - a tool for merging capture files, especially PCAPng files with more than one interface [22]. Wireshark also allows for merging multiple PCAPs, but compared to Tracewrangler, it does not give us the option to tag them.

3.2 Loading the data to database

The capture files are not easy to analyze with Python in its raw form (.pcap, .pcapng or .blf), as every operation requires traversing the whole log file. Furthermore, the application is not supposed to provide information about every packet individually but to provide information about the devices on the network, what ports and protocols they use, and with which other devices they communicate. Therefore, there are many useless data in the raw log file for the purpose of the analyzer, which would make the analyzer unnecessarily slow.

There are multiple Python libraries that can be used for analysis of network capture files, most popular are dpkt, Scapy and Pyshark:

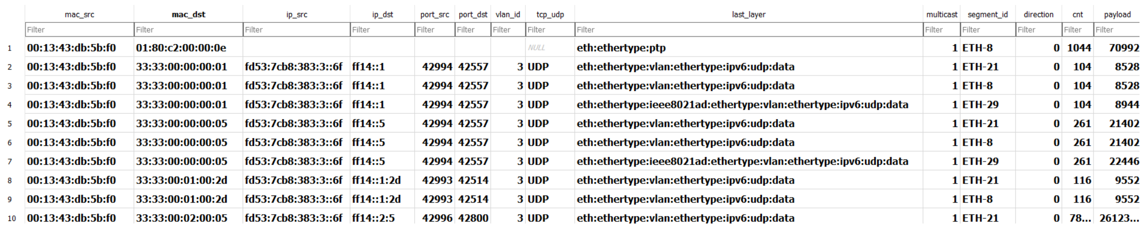
- dpkt is a Python library specifically designed for low-level packet parsing and manipulation. It provides functionality for parsing and dissecting various network protocols, supports packet creation and manipulation and is designed for high performance packet processing.
- Scapy is an packet manipulation library, that is built mainly toward online analysis of the network. Scapy allows the user to create, sniff, modify and send packets. Previously mentioned dpkt can only analyse packets and create them. To send them, raw sockets are needed.
- PyShark is a Python wrapper for Wireshark's packet dissection capabilities. It provides the Python interface to access the same packet details (protocols and metadata) as are available in Wireshark

As Wireshark is basically an industry standard for network traffic analysis, and most people working with networks are comfortable with the tool, PyShark was selected to read and parse the log files. To analyze conversations, nine attributes were identified that need to be extracted from the log file: source and destination MAC address, source and destination IP address, source and destination port, VLAN ID, protocols, and segment ID.

The log parser goes through each packet in the log and extracts the aforementioned attributes to a tuple called record. The size of each packet is extracted separately. All unique records are stored in a Python dictionary as keys, with the total payload stored as values. For each occurrence of record in the dictionary, the payload is updated with the size of current packet as shown below.

```
conversations = dict()
for packet in capture_file:
    record = get_packet_attributes(packet)
    packet_size = get_packet_size(packet)
    if record not in conversations:
        conversations[record] = [1, packet_size]
    else:
        conversations[record][0] += 1
        conversations[record][1] += packet_size
```

Once the capture file is loaded into dictionary, every item (key-value pair) is inserted into the database. This leaves us with much smaller database file (several kB instead of MB) compared to the original capture file. The resulting data stored in the database are shown in image below.



	mac_src	mac_dst	ip_src	ip_dst	port_src	port_dst	vlan_id	tcp_udp	last_layer	multicast	segment_id	direction	cnt	payload
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	00:13:43:db:5b:f0	01:80:c2:00:00:0e						NULL	eth:ethertype:ptp	1	ETH-8	0	1044	70992
2	00:13:43:db:5b:f0	33:33:00:00:00:01	fd53:7cb8:383:3::6f	ff14::1	42994	42557	3	UDP	eth:ethertype:vlan:ethertype:ipv6:udp:data	1	ETH-21	0	104	8528
3	00:13:43:db:5b:f0	33:33:00:00:00:01	fd53:7cb8:383:3::6f	ff14::1	42994	42557	3	UDP	eth:ethertype:vlan:ethertype:ipv6:udp:data	1	ETH-8	0	104	8528
4	00:13:43:db:5b:f0	33:33:00:00:00:01	fd53:7cb8:383:3::6f	ff14::1	42994	42557	3	UDP	eth:ethertype:ieee8021ad:ethertype:vlan:ethertype:ipv6:udp:data	1	ETH-29	0	104	8944
5	00:13:43:db:5b:f0	33:33:00:00:00:05	fd53:7cb8:383:3::6f	ff14::5	42994	42557	3	UDP	eth:ethertype:vlan:ethertype:ipv6:udp:data	1	ETH-21	0	261	21402
6	00:13:43:db:5b:f0	33:33:00:00:00:05	fd53:7cb8:383:3::6f	ff14::5	42994	42557	3	UDP	eth:ethertype:vlan:ethertype:ipv6:udp:data	1	ETH-8	0	261	21402
7	00:13:43:db:5b:f0	33:33:00:00:00:05	fd53:7cb8:383:3::6f	ff14::5	42994	42557	3	UDP	eth:ethertype:ieee8021ad:ethertype:vlan:ethertype:ipv6:udp:data	1	ETH-29	0	261	22446
8	00:13:43:db:5b:f0	33:33:00:01:00:2d	fd53:7cb8:383:3::6f	ff14::1:2d	42993	42514	3	UDP	eth:ethertype:vlan:ethertype:ipv6:udp:data	1	ETH-21	0	116	9552
9	00:13:43:db:5b:f0	33:33:00:01:00:2d	fd53:7cb8:383:3::6f	ff14::1:2d	42993	42514	3	UDP	eth:ethertype:vlan:ethertype:ipv6:udp:data	1	ETH-8	0	116	9552
10	00:13:43:db:5b:f0	33:33:00:02:00:05	fd53:7cb8:383:3::6f	ff14::2:5	42996	42800	3	UDP	eth:ethertype:vlan:ethertype:ipv6:udp:data	1	ETH-21	0	78...	26123...

Figure 3.1. Data stored in the database.

This solution works fine for smaller capture files in size of a few megabytes. However, for bigger log files in the order of gigabytes, the approach of reading packets one by one with PyShark is not sufficient because even reading all packets without manipulation takes several hours.

As the logs measured on a car provided to us were too large for Python processing, another way of processing logs was used. The log file is first exported to a csv file using TShark, which is a command-line interface version of a Wireshark.

```
tshark -r filename.blf -T fields \
-e eth.src -e eth.dst \
-e ip.src -e ip.dst \
-e ipv6.src -e ipv6.dst \
-e udp.srcport -e udp.dstport \
-e tcp.srcport -e tcp.dstport \
-e frame.len -e frame.protocols \
-e vlan.id -e frame.interface_name \
| sort > output.csv
```

Once the extraction is finished, the csv file is read line by line and stored in the database the same way as when the packets are read with Python library, resulting in the same database, so both options are compatible with the rest of the app. Using tshark directly without the Python wrapper is much faster.

Chapter 4
Offline Topology Mapping

This chapter explains the process of creating a network topology graph from packets sniffed at multiple locations in the network. The goal is to identify nodes, L2 switches, and L3 routers in the network and give the user information on how these devices are connected. The first section details how the topology is created from packets captured on a single location on the network and presents its limitations. The second section describes the implementation of an algorithm to mitigate these limitations by combining the information captured in different segments. The last section analyzes the resulting topology reconstructed from packets captured on a vehicle.

4.1 Topology structure

Algorithms described in the following chapters are implemented in the library analyzer. The network topology graphs created by these algorithms are constructed using the Python igraph library.

The super-class GraphBase initializes instance of an igraph Graph, that contains detected network topology. The super-class further defines graph manipulation methods that are common for every graph, such as getting vertex by IP or MAC address, adding and copying vertices and edges, and plotting the graph.

The class SegmentGraph inherits from the super-class GraphBase and defines methods for preprocessing the data as described in section 4.2 and for building the segment graph as described in Section 4.3.

The class NetworkGraph inherits from the super-class GraphBase and defines methods for merging multiple segment graphs into one graph as described in section 4.4. The code snippets provided for this class do not exactly match the code in the library. They contain only the necessary information to understand the algorithm (some graph manipulation steps were removed from the text to make snippets shorter). As this class works both with the graph from the SegmentGraph class and with graph from the NetworkGraph itself, the ng_ prefix in the code refers to vertices from NetworkGraph and the sg_ prefix refers to vertices from SegmentGraph.

Each device in the graph is represented by an igraph Vertex. In the current implementation, four attributes are assigned to each vertex, namely device_type, mac_address, ip_address, and vlan. If some more information needs to be added for the devices in the future, this can be easily done by adding a new attribute to the vertex in the add_vertex method of the Graph-Base.

Devices in the graph are connected by an igraph Edge, where each edge holds the following attributes: name, label, color, and width.

4.2 Data preprocessing

To create a graph from a single segment, the unique sets of source IP address, destination IP address, source MAC address and destination MAC address captured on one interface and in one direction are queried from the database. Next, the addresses are divided into two groups according to their direction, such that one group contains only pairs of source IP address and MAC address, and the second group contains destination pairs. These two groups represent two sets of physically connected devices at each end of the measured segment.

The packets queried from the database must be unidirectional, meaning that every MAC address and IP address in the set queried from the database is only present as a source address or only as a destination address. This gives information on which side of the measured segment the device with the given address is located. From this point on, it is possible to build a segment graph (section 4.3.1).

However, the packets captured on the same interface but in opposite direction are also available in the database. One way would be to repeat the steps from the first paragraph, build a second segment graph, and merge these graphs later. This solution is more computationally demanding, as the number of segment graphs to merge is doubled. Instead, the address pairs from the second direction are also queried from the database and added to the groups, this time with little difference. In this case, the destination addresses are added to the group with the source addresses, and vice versa, so the devices are added to the correct side of the measured segment. Another benefit of merging information from the same segment now is that it includes devices that send only multicast packets. A device that sends only multicast packets gives no information about which device is on the other side of the measured segment, as at this moment, there is no information about which physical device belongs to which multicast group. When the measurements in both directions from the same segment are merged now, it guarantees at least one device at each end of the measured segment, as the source address in each direction cannot be multicast.

4.3 Single segment topology creation

In Section 4.2, the addresses captured on one segment were divided into two groups based on which side of the measured segment they physically are. This step adds and connects the vertices that represent the physical devices to the segment topology graph. Based on the working principle of an L2 switch and an L3 router, four distinct types of topology that can be detected at each end of the measured segment were identified. Each case is uniquely recognized by the number of distinct IP and MAC addresses in the group, as explained in the following section.

4.3.1 Single segment topology principle

In the first case, only a single IP address and a single MAC address are present in the group. The graph element will contain only one vertex, representing the end device (node).

```
def create_graph_element(graph, macs, ips):
    m, i = len(set(macs)), len(set(ips))
    if i == 1:  # 1 node vertex
        origin = graph.add_vertex(NODE, macs[0], ips[0])
```

In the second case, suppose that the group contains n distinct IP addresses and a single unique MAC address. In that case, the graph element will contain one router vertex with the corresponding MAC address and n unknown vertices with the corresponding IP addresses that are connected to the router vertex. An unknown vertex is used to express that there is insufficient information about the device communicating through the router. For example, whether it is connected directly to the router or whether the devices behind it (from the point of measurement) are connected through a switch and then through a router.

```
    elif m == 1:  # 1 router and i unknown vertices
        origin = graph.add_vertex(ROUTER, macs[0])
        for ip in ips:
            neighbor = graph.add_vertex(UNKNOWN, ip)
            graph.add_edge(origin, neighbor)
```

In the third case the group contains multiple distinct IP addresses and multiple distinct MAC addresses, but its count n is equal. The graph element consists of the switch vertex and n node vertices with the corresponding MAC and IP addresses connected to the switch vertex.

```
    elif m == i:  # 1 switch and m node vertices
        origin = graph.add_vertex(SWITCH)
        for mac, ip in zip(macs, ips):
            neighbor = graph.add_vertex(NODE, mac, ip)
            graph.add_edge(origin, neighbor)
```

In the last case, the group contains n different IP addresses and m distinct MAC addresses where $n > m$. If a specific MAC address is represented in the graph only once, with the corresponding IP address, a node with these addresses is added to a graph. Every successive node is connected to this node by a switch. If the same MAC address is present in the group multiple times, each time with different IP address, a router with this MAC address is added to a graph and connected to the switch. For every IP address corresponding to this MAC address, an unknown node is added to the graph and connected to this router.

```
    else:
        origin = graph.add_vertex(SWITCH)
        detected_router_macs = []
        for idx, mac in enumerate(macs):
            if macs.count(mac) == 1:  # mac belongs to node
                neighbor = graph.add_vertex(NODE, mac, ips[idx])
                graph.add_edge(origin, neigbor)
            else:  # mac belongs to router
                if mac in detected_routers_macs:
                    continue
                detected_routers_macs.append(mac)
                router = graph.add_vertex(ROUTER, mac)
                graph.add_edge(origin, router)
                for i in range(idx, len(macs)):
                    if macs[i] != mac:
                        continue
                    unknown = graph.add_vertex(UNKNOWN, ips[i])
                    graph.add_edge(router, unknown)
    return origin
```

After creating the graph elements for each of the two groups of addresses, the origin vertices of each graph element are connected by an edge. This edge will be referred to as a measured edge and is marked by a wide line in the graph.

4.3.2 Single segment topology example

To better understand the rules presented in the previous section, this section provides analysis of the resulting segment graphs built from packets captured in a simulated network shown in the picture 4.1 with eight end devices, three switches and one router. To demonstrate the first and the last rule from Section 4.3.1 the communication was set such that NODE1.1 sends packets to every end device in the network, and the packets were captured at the link *eth1.1*. The addresses extracted from the capture file are shown in Table 4.1. When these addresses are divided into two groups by direction, the source addresses contain only one unique MAC address (00:50:79:66:68:00) and IP address (10.1.1.1), so on one side of the measured edge, there should be only one node. The destination addresses contain 7 unique pairs of MAC addresses and IP addresses, but only 4 unique MAC addresses. This means that the second side of the measured segment will contain 1 switch with 3 devices, and the switch will also be connected to a router that is further connected to 4 unknown vertices as shown in the picture 4.2

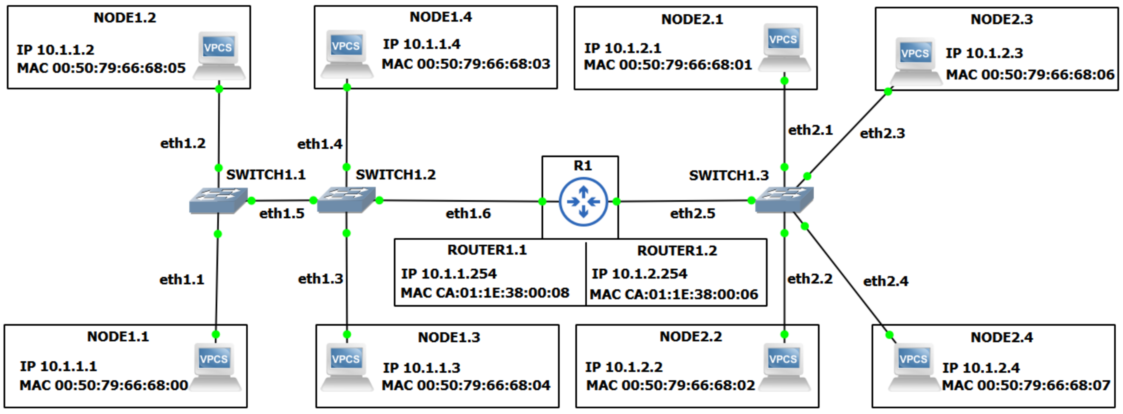


Figure 4.1. Simulated network topology.

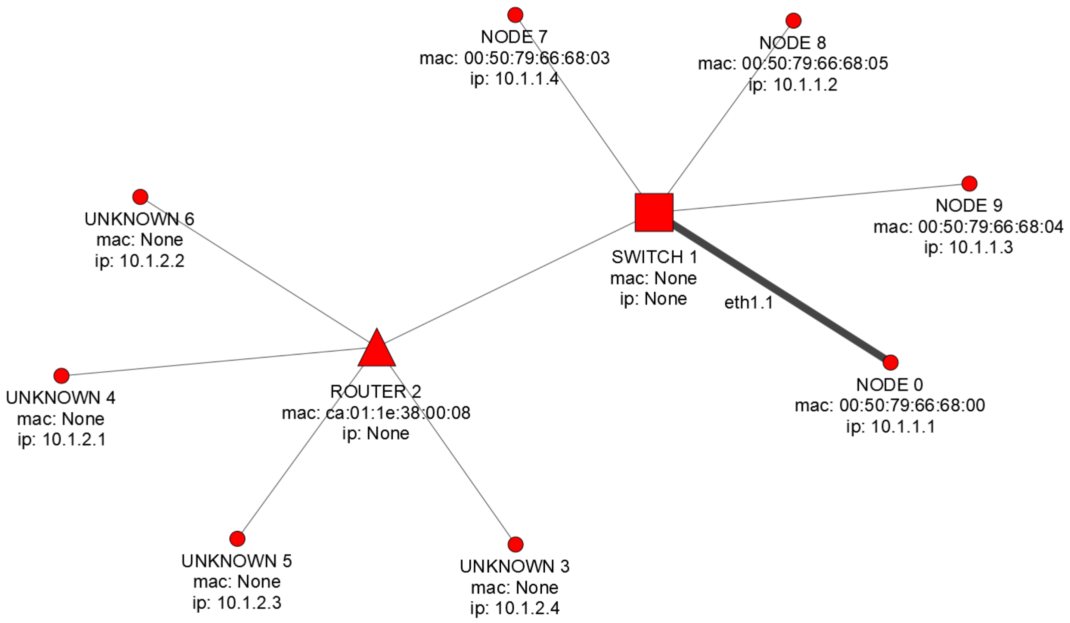


Figure 4.2. Segment graph constructed from packets captured at link *eth1.1*.

Source MAC	Destination MAC	Source IP	Destination IP
00:50:79:66:68:00	00:50:79:66:68:05	10.1.1.1	10.1.1.2
00:50:79:66:68:00	00:50:79:66:68:04	10.1.1.1	10.1.1.3
00:50:79:66:68:00	00:50:79:66:68:03	10.1.1.1	10.1.1.4
00:50:79:66:68:00	ca:01:1e:38:00:08	10.1.1.1	10.1.2.1
00:50:79:66:68:00	ca:01:1e:38:00:08	10.1.1.1	10.1.2.2
00:50:79:66:68:00	ca:01:1e:38:00:08	10.1.1.1	10.1.2.3
00:50:79:66:68:00	ca:01:1e:38:00:08	10.1.1.1	10.1.2.4

Table 4.1. Captured addresses at link *eth1.1*

Source device	Target device
NODE2.1	NODE1.1
NODE2.2	NODE1.1
NODE2.3	NODE1.2
NODE2.4	NODE1.2

Table 4.2. Setup of communication to demonstrate second and third case from the Section 4.3.1.

To demonstrate the second and third rules from Section 4.3.1, the communication was set as shown in Table 4.2, and the packets were captured at the link *eth2.5*. In this situation, the captured addresses are shown in the Table 4.3. The source addresses contain 4 unique MAC addresses and 4 unique IP addresses. This means that one side of the measured segment consists of one switch and four end devices. Destination addresses contain only one unique MAC address and two unique IP addresses, resulting in a router with the captured MAC address connected to two unknown vertices with the captured IP addresses. Figure 4.3 shows the resulting segment graph.

Source MAC	Destination MAC	Source IP	Destination IP
00:50:79:66:68:01	ca:01:1e:38:00:06	10.1.2.1	10.1.1.1
00:50:79:66:68:02	ca:01:1e:38:00:06	10.1.2.2	10.1.1.1
00:50:79:66:68:06	ca:01:1e:38:00:06	10.1.2.3	10.1.1.2
00:50:79:66:68:07	ca:01:1e:38:00:06	10.1.2.4	10.1.1.2

Table 4.3. Captured addresses at link *eth2.5*

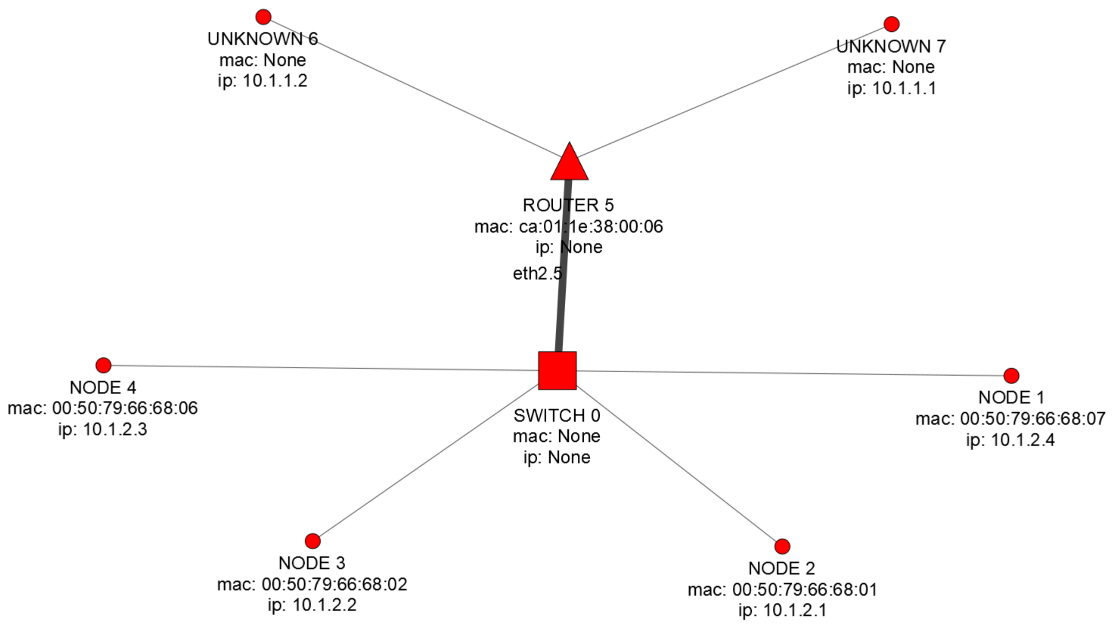


Figure 4.3. Segment graph constructed from packets captured at link *eth2.5*.

4.3.3 Single segment topology limitations

When the segment topologies created by the algorithm shown in Figures 4.2 and 4.3 are compared to the original topology of the network shown in Figure 4.1, several differences may be observed. These differences are caused by the fact that the packets measured in a single segment do not provide enough information for the algorithm to reconstruct the topology of the entire network. The identified limitations of this algorithm are listed below:

- The first obvious limitation is that only devices communicating on a measured segment can be seen by the algorithm. An example is shown in figure 4.3 where only 6 nodes or unknown nodes were detected, as other devices were not communicating on the measured segment (see Table 4.2).
- The second limitation is that it is impossible to recognize multiple switches connected in a cascade at the ends of a segment; they will always look like a single switch. This limitation can be seen in the picture 4.2, where nodes 0, 7, 8, and 9 seem to be connected to a single switch, but when in the original network, the devices with the corresponding addresses are connected to two distinct switches.
- The third limitation is that there is not enough information about the connection of devices behind the router, whether they are connected directly to the detected router, through a switch, or there is another router in a way. This is visible in both pictures 4.2 and 4.3 where the unknown vertices

are always connected directly to a router, but in the original network, the devices with corresponding IP addresses are connected through a switch that is not detected.

- The fourth limitation is that if only a single end device communicates through a router, the router cannot be detected properly. The router will be incorrectly detected as an end device with a router's MAC address and end-device IP address, as in this case, the difference between unique IP addresses and unique MAC addresses cannot be detected. An example of this limitation can be seen in the topology shown in the picture 4.4. The vertex with the name `NODE 4` is detected as an end device with the IP address of NODE2.3 from the original network but has the MAC address of a router.
- The last limitation is that the router's IP address cannot be detected. The current algorithm will make it look like another unknown device is in the network. This limitation can be mitigated by using only packets that use transport protocol for the algorithm, as the TCP or UDP packets usually do not have origin at the router.

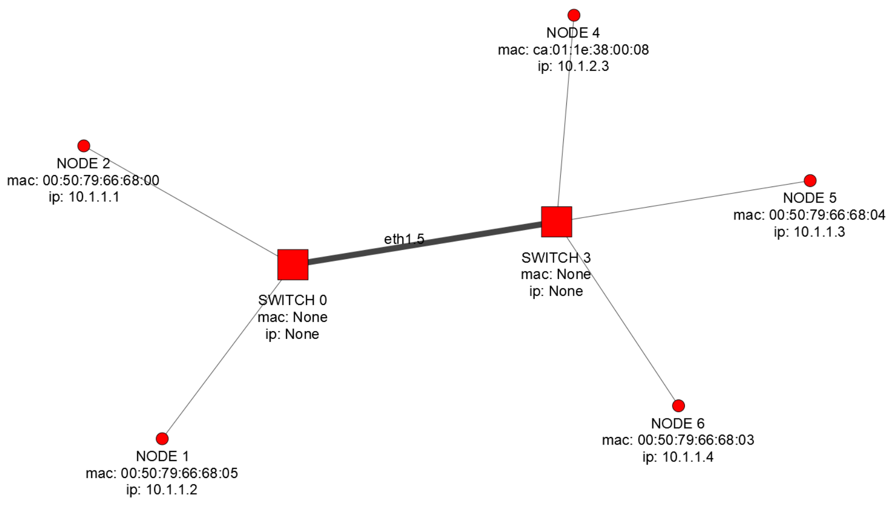


Figure 4.4. Segment graph constructed from packets captured at link *eth1.5*.

Source device	Target device
NODE1.1	NODE1.4
NODE1.2	NODE1.3
NODE1.2	NODE2.3

Table 4.4. Setup of communication to demonstrate limitations switch switch *eth1.5*.

4.4 Network graph

As explained in the previous section, the topology created from a measurement on a single network segment has some limitations. However, the informative value of the created topology can be improved by combining information from multiple measured segments. Unless every segment is measured and unless it is ensured that every device on the network communicates, there is still some uncertainty in the topology; therefore, the task is to extract as much information as possible regarding the limitations stated in the section above.

In the first step, one of the segment graphs is used as a base for the network graph. The algorithm then iterates through all other segment graphs that were created and updates the additional information to the network graph, as explained below.

4.4.1 Localization and updating the devices at the end of the measured edge

The algorithm takes the devices at the end of the measured edge in the segment graphs and tries to locate them in the network graph. Three distinct cases may occur depending on the type of device, as explained in the following list. There cannot be an unknown device type at the end of the measured edge by the design of the segment graph algorithm.

- The algorithm searches for the node (end device) - it checks if there is a vertex with the same IP address in the network graph. If there is no such node, a new vertex is added to the graph, and the MAC and IP addresses are copied from the searched vertex. If there is such a vertex and its device type is unknown, the device type is changed to node, and the MAC address is copied from the searched vertex. If there is such a vertex whose device type is a node, the MAC addresses of the searched vertex and the found vertex must be compared. If they are different, it means that one of the MAC addresses belongs to the router (as explained in Section 4.3.3). In this case, the device type of found node is changed to a router, and its IP address is set to None, as the router IP addresses are not detected by the algorithm. A new vertex with the device type node is added to the graph, and the IP address of the searched vertex is copied to this vertex. Algorithm updates both MAC addresses for both vertices until it can reliably tell which one belongs to which vertex (this information may be available in other segment graphs). If the MAC addresses are the same, vertex parameters remain unchanged.

```
def find_node(ng_graph, sg_node):
    found = ng_graph.get_device_by_ip(sg_node.get_ip())
    if found:
        if found.is_unknown():
            found.set_type(NODE)
            found.set_mac(sg_node.get_mac)
    else:
        found = ng_graph.get_device_by_mac(sg_node.get_mac())
        if found and found.is_router():
            new_node = ng_graph.add_vertex(UNKNOWN, sg_node.get_ip())
            ng_graph.add_edge(found, new_node)
        else:
            found = ng_graph.copy_vertex(sg_node)
    return found
```

- The algorithm searches for the router - it checks if there is a vertex with the same MAC address in the network graph. If there is no such vertex, a new vertex with the device-type router is added to the graph, and the MAC address is copied from the searched vertex. If there is only one vertex found in the network graph and its device type is a node, its device type will be changed to a router. Next, the IP address of the found vertex is copied and removed, and a new unknown vertex with the said IP address is added to the network graph and connected to the found vertex. This step fixes the fourth limitation of a segment graph (Section 4.3.3). If there is only one vertex with the corresponding MAC address and its device type is a router, the parameters remain unchanged.

```
def find_router(ng_graph, sg_router):
    found = ng_graph.get_device_by_mac(vertex.get_mac())
    if not found:
        found = ng_graph.copy_vertex(sg_router)
    if device_found.is_node():
        new_node = ng_graph.add_vertex(UNKNOWN, found.get_ip())
        ound.set_device_type(ROUTER)
        ng_graph.add_edge(found, new_node)
    return device_found
```

- The algorithm searches for the switch - a switch cannot be uniquely identified by an IP address or MAC address. Therefore, a different approach is used that requires iterating through all the switch vertices in the network graph. For each switch vertex in the network graph, its neighbors are compared to the neighbors of the searched switch vertex. If the switch vertex in the network graph is connected to any end device with the same MAC address as the switch in the segment graph is linked to, this vertex is considered the found vertex. This approach may return two switches that satisfy this condition (as a result of the second limitation in Section 4.3.3). This situation can be seen when using the segment graph from Figure 4.4 as the base for the network graph and attaching the segment graph from Figure 4.2. In this case, the algorithm will return both `SWITCH 0` and `SWITCH 3` as found vertices when searching for `SWITCH 0` from the segment graph. In this case, the node at the other end of the measured edge in the segment graph needs to be considered with higher priority. In the example, there is `NODE 0` with the MAC address 00:50:79:66:68:00 at the other end of the measured edge (Figure 4.2), and this node is connected to `SWITCH 0` in the network graph 4.4. Therefore, only `SWITCH 0` will be considered the found vertex in this case. And once again, if no such switch exists in the network graph, a new switch vertex is added to the graph.

```
def find_switch(ng_graph, sg_switch, sg_ancestor_mac):
    ng_switches = graph.vs.select(device_type=SWITCH)
    if not ng_switches:
        found_switch = ng_graph.add_vertex(SWITCH)
        return found_switch
    sg_switch_neighbors = [neighbor.get_mac() \
        for neighbor in sg_switch.neighbors()]
    found_switch = None
    for ng_switch in ng_switches:
        ng_switch_neighbors = [neighbor.get_mac() \
            for neighbor in ng_switch.neighbors()]
        if set(ng_switch_neighbors) & set(sg_switch_neighbors):
            if sg_ancestor_mac in ng_switch_neighbors:
                return ng_switch
            found_switch = ng_switch
    if not found_switch:
        found_switch = ng_graph.add_vertex(SWITCH)
    return found_switch
```

4.4.2 Updating the edge between the devices

In the next step, the algorithm checks if the two vertices found in the previous step are connected. If they are connected, it only marks the edge as measured; otherwise, it adds a new edge and marks it as measured. There is one more special case to handle: when both devices from the segment graph are pointing to the same device from the network graph. This situation will occur when the segment graph that should be added to the network graph is measured between two switches. In the network graph these switches are currently represented only as a single switch (due to the second limitation from Section 4.3.3). In this case, another new switch vertex is added to the network graph and connected to the switch that was found - this edge is marked as the measured edge.

4.4.3 Updating remaining devices from the segment graph into the network graph

The last step is to iterate through the neighbors for each device at the end of the measured edge from the segment graph, compare them to neighbors of the found device in the network graph, and possibly update these devices and connections. In further text, the device on the other end of the measured edge is not considered a neighbor, as this neighbor was resolved separately in the previous step. As a result, if the device at the end of the measured edge is a node (end device), no changes are needed, because this device is connected only to the device that is at the other end of the measured edge (for example, NODE 0 is connected only to SWITCH 1 in the segment graph 4.2).

If the device at the end of the measured edge is a router, its neighbors will be only unknown vertices. In this case, the algorithm only checks if the device is present in the network graph (it searches by IP address). If the device is present, no updates are made to the network graph, as there is no new information due to the third limitation from Section 4.3.3). If a vertex with given IP address is not present in the network graph, a new unknown vertex with the corresponding IP address is added to the graph and connected to the router.

```
def update_router_neighbors(ng_graph, ng_router, sg_router, \
                            sg_ancestor):
    edges = set()
    for sg_neighbor in sg_router.neighbors():
        if sg_neighbor == sg_ancestor:
            continue
        ng_neighbor = ng_graph.get_vertex_by_ip(sg_neighbor)
        if ng_neighbor is None:
            ng_neighbor = ng_graph.copy_vertex(sg_neighbor)
            ng_graph.add_edge(ng_router, ng_neighbor)
        elif ng_neigbor.is_unknown() and sg_ancestor.is_router():
            ng_neigbor.delete_edges()
            ng_graph.add_edge(ng_neigbor, ng_router)
```

If the device at the end of the measured edge is a switch, it can have only two types of neighbors - node or router. It cannot be an unknown vertex, as those are connected exclusively to a router. It also cannot be a switch because if a switch were neighboring in the real network, from the measurement point of view, it would be seen as only a single switch. The two situations that can occur for the switch neighbor based on the device type are explained below.

- If the searched neighbor is a node vertex and it is already present in the network graph as a node vertex, no changes are made to the network graph. There is one exception to this rule: When both devices at the ends of the measured edge are switches, the found neighbor may need to be reconnected to the appropriate switch, as this is the only situation where the algorithm knows the correct connection. If no corresponding vertex is found, a new node vertex is added and connected to the switch. If a corresponding unknown vertex is found, the graph is updated with the additional information. The device type of the vertex is changed to node, and the missing MAC address is copied from the corresponding vertex in segment graph. Then the connection of the updated vertex is deleted, as it was previously connected to a router, but now it is clear that this vertex should be connected to a switch. Then a new connection from the switch vertex to the node vertex is added. Finally, it is also checked that the switch itself is connected to the router that was disconnected from the unknown vertex. It is known that this vertex was communicating through this router, therefore there must be a path. If it is missing, a new edge from the switch to the said router is added to the network graph.

- If the searched neighbor is a router, it may be found on the network graph as a router or as a node. If it is found as a router, no changes to this vertex are made. If it is found as a node, its device type is changed to a router, and its IP address is deleted. A new unknown vertex with the IP address that was removed from the router vertex is added to the network graph and connected to the router. If no corresponding vertex is found in the network graph, a new vertex, a copy of the searched router vertex, is added to the network graph. As a last step, all neighbors of a searched router from the segment graph are searched if they are already present in the network graph. As all neighbors of a router vertex will be unknown vertices (except for the device on the other side of the measured edge) if the routers neighbors are present in a network graph, no changes are made here; only if it is not present, the unknown vertex is copied to the network graph and connected to the router.

4.5 Network topology creation example

In this section, a concrete example of reconstructing a network graph from multiple segment graphs is presented and compared to the original network. The algorithm will be used to reconstruct the topology of the network shown in Figure 4.1, using the segment graph shown in Figure 4.2 as the base for the network and adding information from the segment graph shown in Figures 4.3 and 4.4.

4.5.1 Network graph after adding information from the segment graph 4.3

This section describes complete steps that the algorithm is doing when using 4.2 as a base for the network graph and adding information from the segment graph in Figure 4.3.

First, it checks if the devices at the end of measured segment are already present in the network graph. In this case, at one end of the measured edge is a router with MAC address ca:01:1e:38:00:06 that is not present in the network graph. Therefore, the searched router vertex is copied to the network graph. On the other side of the measured edge is a switch connected to any of the vertices with MAC addresses 00:50:79:66:68:01, 00:50:79:66:68:02, 00:50:79:66:68:06 or 00:50:79:66:68:07. The only switch currently present in the network graph is connected only to vertices with MAC addresses 00:50:79:66:68:00, 00:50:79:66:68:03, 00:50:79:66:68:04, and 00:50:79:66:68:05. As the MAC addresses of the neighbors of the searched switch and the neighbors of the switch in the network graph do not overlap, a new switch vertex is also added to the network graph. The newly created vertices are connected by an edge that is marked as the measured edge. The graph after this step is shown in figure 4.5.

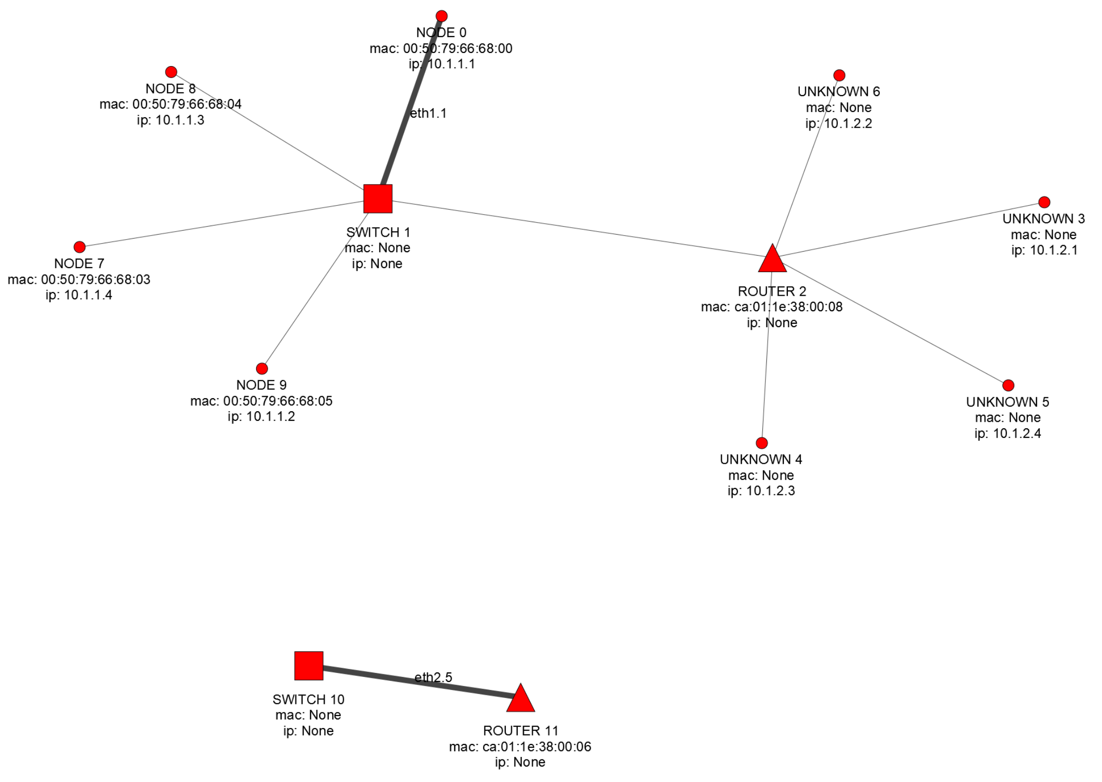


Figure 4.5. Change in network topology after first two steps.

Next, the neighbors of the switch in the segment graph are searched in the network graph (except for the router, as this is a device on the end of a measured edge that is handled separately). These neighbors are nodes with IP addresses 10.1.1.1, 10.1.1.2, 10.1.1.3, and 10.1.1.4. The vertices with these IP addresses are already present in the network graph, although their device type is unknown, and they are connected to the router with MAC address ca:01:1e:38:00:08, which based on the current segment graph is incorrect. The device type of these vertices is changed to node, and the corresponding MAC address is copied from the segment graph. Then the connections are fixed. First, the connection to the router with MAC address ca:01:1e:38:00:08 is deleted for each vertex, and then they are connected to the newly created switch. After creating new connections between nodes and switch, the algorithm checks if there is a path in the network graph between these nodes and the router to which they were originally connected as unknown vertices. There must be, as these devices were communicating through both of these routers. Because such path does not exist, the router with MAC address ca:01:1e:38:00:08 is connected to the router with MAC address ca:01:1e:38:00:06. This edge is marked with red, signifying that it is unknown what is between these routers. The result after this step is shown in figure 4.6

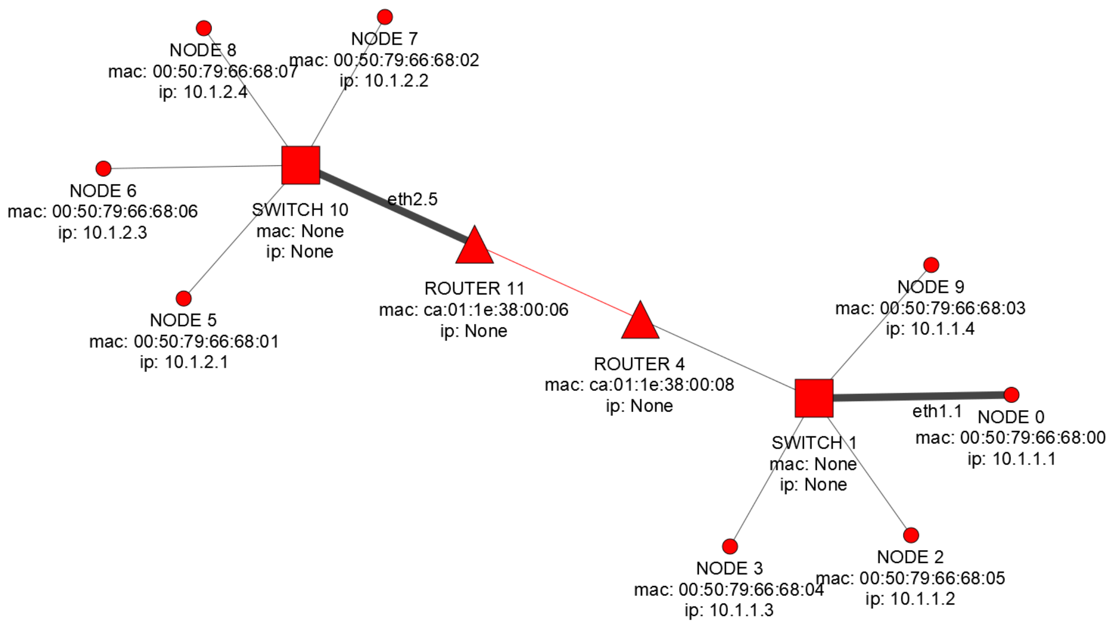


Figure 4.6. Network topology as detected by our algorithm after adding information from graph in figure 4.2 to graph 4.3.

The last remaining step is to check if all the router's neighbors in the segment graph are present in the network graph. In this case, the router has only two neighbors, unknown vertices with IP addresses 10.1.1.1 and 10.1.1.2. Both vertices are already represented in the network graph as nodes. Therefore, there is no new information to add and the network topology will not change.

4.5.2 Network graph after adding information from the segment graph 4.4

This section describes how the current network graph 4.6 changes, when another segment graph 4.4 is added to it.

The same process as in the previous example is repeated again. First, the devices at the ends of the measured edge from the segment graph are localized in the network graph. In this case, the searched devices are two switches, one connected to nodes with MAC addresses 00:50:79:66:68:00 and 00:50:79:66:68:05, and the second connected to nodes with MAC addresses 00:50:79:66:68:03, 00:50:79:66:68:00, and ca:01:1e:38:00:08.

When these switches are located in the network graph as explained in chapter 4.4.2, both searched vertices will point to the same vertex in the network graph 4.6, SWITCH 1. As the second switch is missing in the network graph, one more switch vertex (SWITCH 12 in the network graph 4.7) is added and connected to SWITCH 1 by an edge. This edge will be marked as a measured edge.

The neighbors are updated for each switch. SWITCH 0 of the segment graph 4.4 is connected only to nodes with MAC addresses 00:50:79:66:68:00 and 00:50:79:66:68:05. SWITCH 1 in the network graph 4.6 is already connected to these vertices. Therefore, no changes are made to the graph in this step. The second switch (SWITCH 3) from the segment graph 4.4 is connected to nodes with MAC addresses 00:50:79:66:68:03, 00:50:79:66:68:00, and ca:01:1e:38:00:08. The vertices with MAC addresses 00:50:79:66:68:03 (NODE 9), 00:50:79:66:68:04 (NODE 3) are located on the network graph 4.6 and reconnected from SWITCH 1 to the newly added SWITCH 12. The vertex with MAC address ca:01:1e:38:00:08 (ROUTER 4) is also located and reconnected in the same way. But for this vertex, the algorithm will find that the IP address and device type of the searched vertex and the located vertex do not match (10.1.2.3 != None and ROUTER != NODE). This means that the searched vertex was incorrectly detected in the segment graph due to the fourth limitation mentioned in Section 4.3.3, and that there should be a vertex with this IP address (10.1.2.3) in the network graph as well. When this IP is searched in the network graph, it is already present as NODE 6, so no further changes are needed. If it was not present, the algorithm would create a new unknown vertex with given IP address and connect it to the ROUTER 4.

This process will lead to the final reconstructed topology of the network as shown in figure 4.7

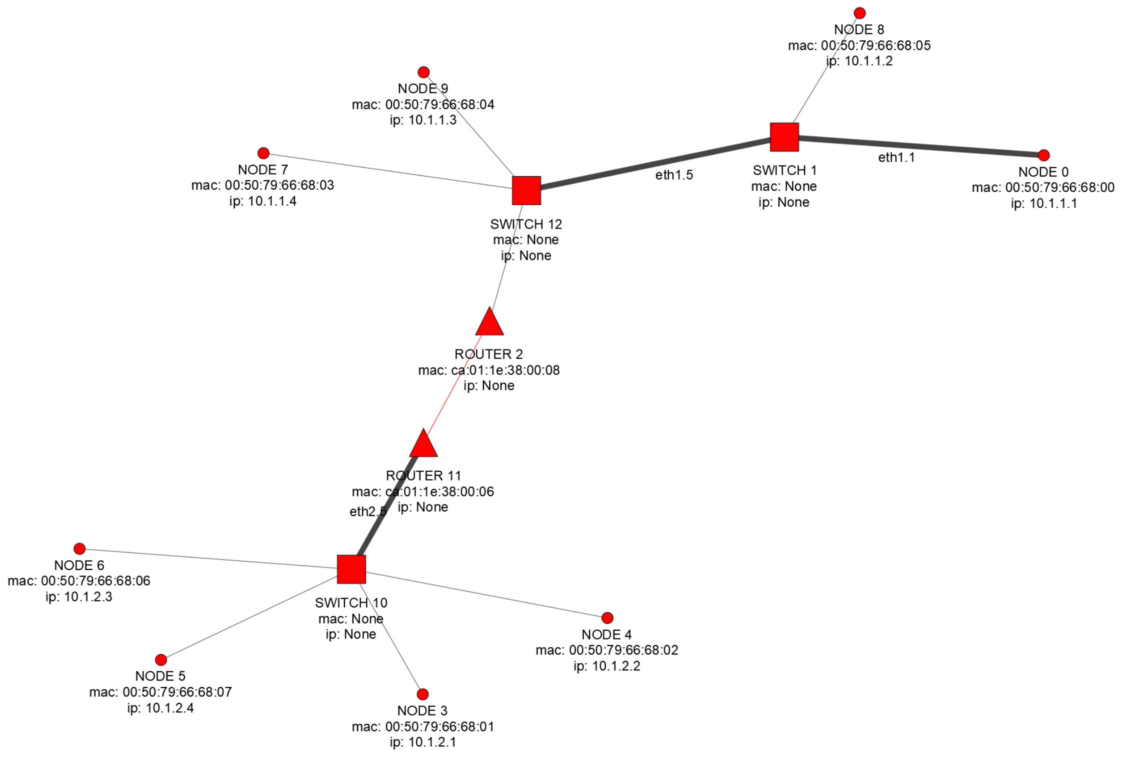


Figure 4.7. Network topology as detected by our algorithm after adding information from graph in figure 4.4

When the reconstructed topology is compared with the original network topology shown in figure 4.1, all eight end devices are detected and they have the correct MAC and IP address. All three switches on the network are detected and connected to the correct devices. The router is represented as two vertices, one for each detected interface, as it is impossible to tell that these two interfaces belong to one physical device with this method, but every detected interface is connected to the correct switch.

In conclusion, with this method, it is possible to reconstruct the network topology from multiple segment graphs if a measurement is available between every pair of switches, every device is correctly represented in any of the measured segments at least once, and the measurement was taken at every LAN at least once.

4.5.3 Results from a vehicle

After testing the algorithm on a simulated network, the data measured in a vehicle became available. The seven individual segment graphs that were used to construct the network graph can be seen in the Appendix A. The resulting topology after running the algorithm can be seen in Figure 4.8.

As the network grew bigger, plotting the graph using the igraph plot function (as was the graphs in the previous chapters) proved to be problematic, mainly because the igraph algorithm for laying out the vertices does not account for the labels, resulting in overlaps and bad readability. Therefore, for visualization purposes, all the following graphs in the text were exported to graphML format and opened in CytoScape, an open-source software platform for visualizing complex networks. The Cytoscape session used for visualization is available in the attachments, and it does not change the structure of the graphs in any way, it only maps the attributes of graph elements assigned by the code to visualization (e.g. if the detected device type is ROUTER, it is mapped to visualize a router symbol).

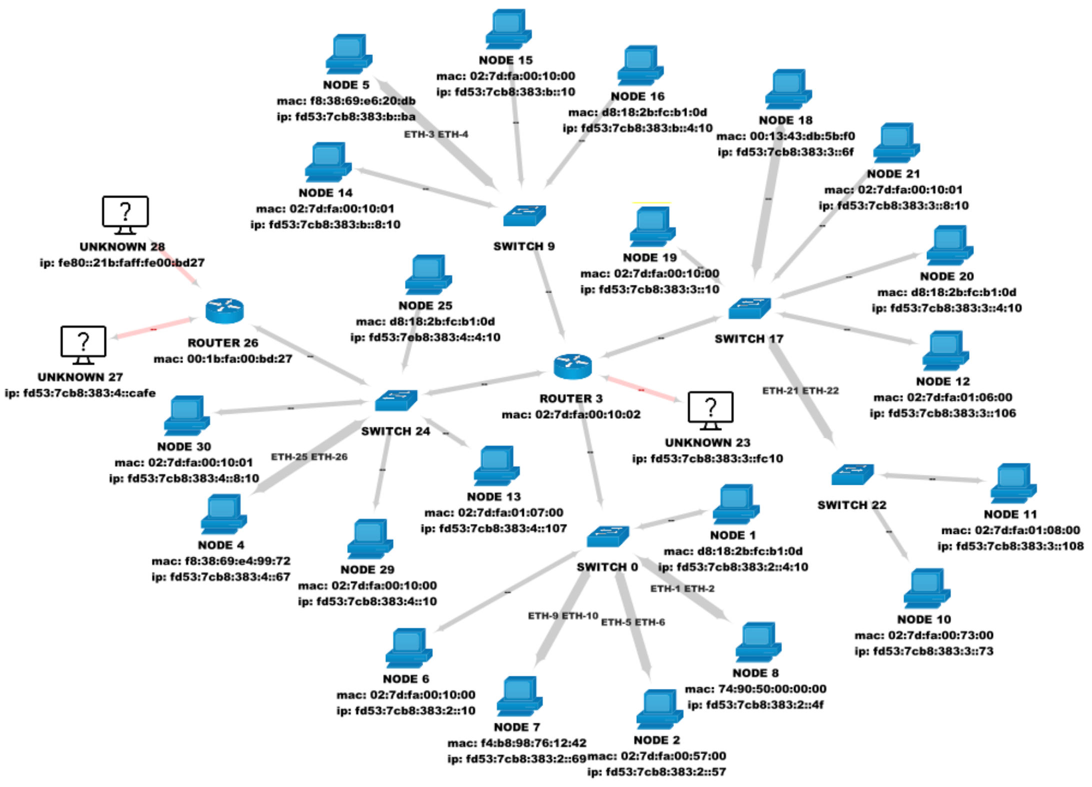


Figure 4.8. Network topology graph of a vehicle as detected by the algorithm.

However, the reconstructed topology graph shown in Figure 4.8 does not represent the topology of the network in a physical sense (that is, how the wires go between the devices). This is caused by the fact that the analyzed in-vehicle network uses VLAN, and some ECUs belong to multiple VLANs (as explained in Section 2.2). In the graph merging algorithm, the presence of the node in the graph was determined by IP address, and because different VLANs use distinct address spaces (not necessarily, but it is a best practice that allows routing between different VLANs), the physical device is represented in the topology multiple times. This can be seen, for example, in nodes 1, 16, 20 and 25, where each node has the same MAC address, but different IP addresses. Another discrepancy is that switches 0, 9, 22 and 24 are connected to the same router interface, which is physically impossible. In reality, it is a single physical switch connected to a router interface by a trunk port. Therefore, to get a physical topology a slightly modified version of algorithm was implemented, that searches nodes by both IP and MAC address and keeps all detected IP addresses for the node with the same MAC address. As the equivalence of the switch is determined by the nodes connected to it, the modification of node-searching algorithm also solves the issue with the same physical switch being present multiple times in the physical topology graph. The result of the modified algorithm is shown in Figure 4.9. The ROUTER 17 and two unknown nodes connected to it are not a part of the in-vehicle network, it is an artifact from the logging computer. This can be removed by introducing an IP address filter during data loading into the database. Otherwise all the addresses that were available in the log file are present in the graph, and the topology resembles the expected topology of the in-vehicle network with VLAN, as was presented in the theoretical part of the thesis Figure 2.3. The comparison with the vehicle documentation showed that switch 0 in Figure 4.9 is in reality not a single switch, but multiple switches connected in cascade. The algorithm could not recognize that due to the limitations stated before.

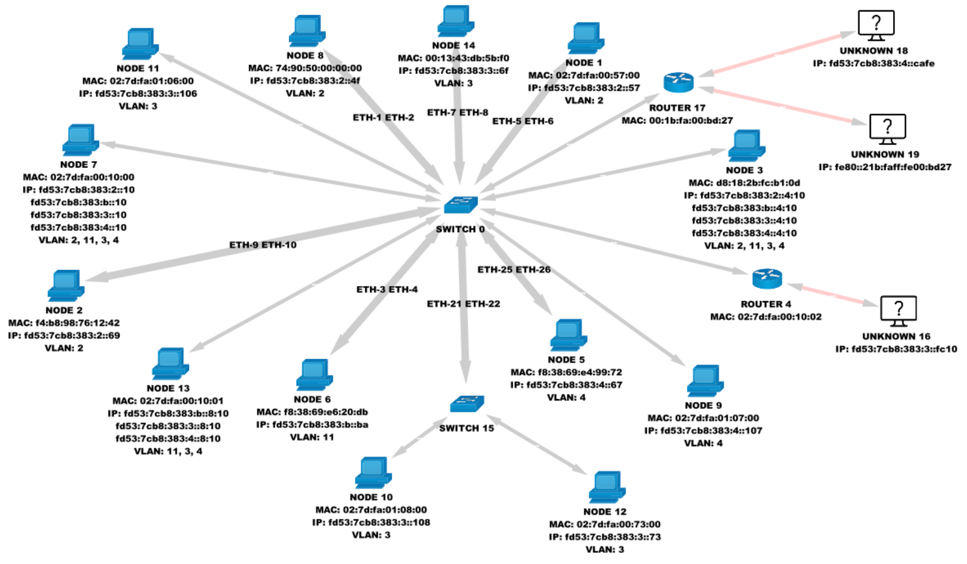


Figure 4.9. Physical network topology graph of a vehicle as detected by the algorithm.

Chapter 5
Traffic Flow Analysis

This chapter explains the process of removing duplicate packets from the database, creating the network flow graph, and network statistics.

5.1 Duplicate packets in capture file

When monitoring and testing automotive networks it is common practice to capture packets from multiple switch ports at once, as explained in chapter 2.4.3. If the network is monitored using this method, the resulting trace may contain the same packets captured multiple times. Let's consider that the multi TAP is connected to the example network from figure 2.2 as shown in figure 5.1.

When Display node sends a packet to every device in the network, the packet sent to Speaker node will be captured two times, as it's first seen travelling from the Display to the Switch and then from the Switch to the Speaker. In comparison, packets travelling to Radio and Console will be seen only once, on their way from the Display to the Switch.

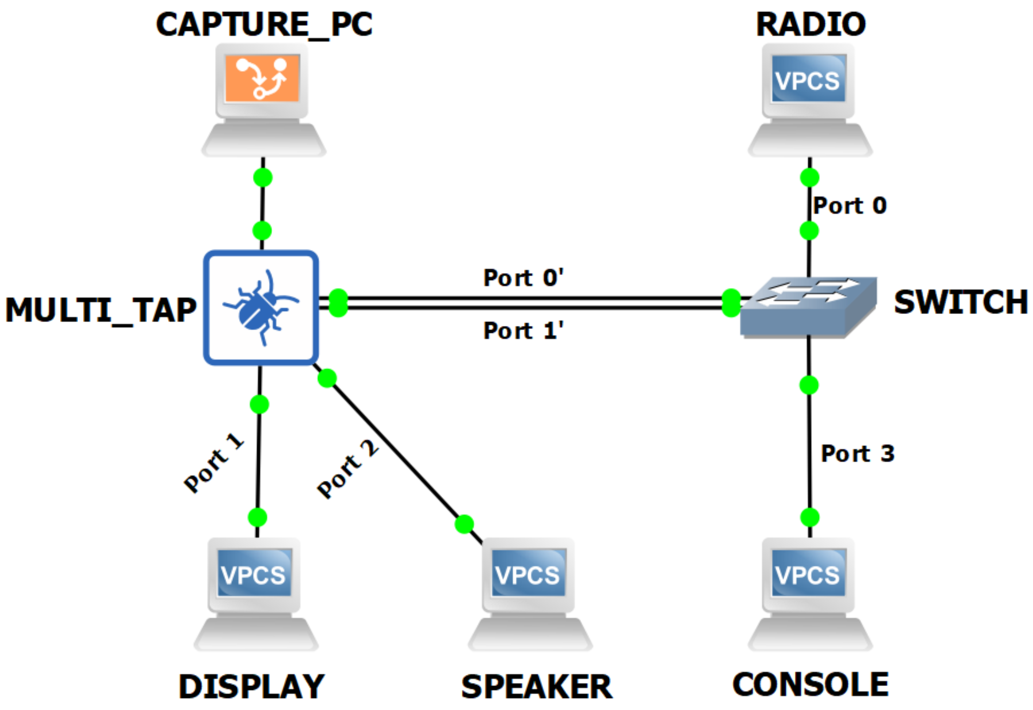


Figure 5.1. Example of multi active TAP connection.

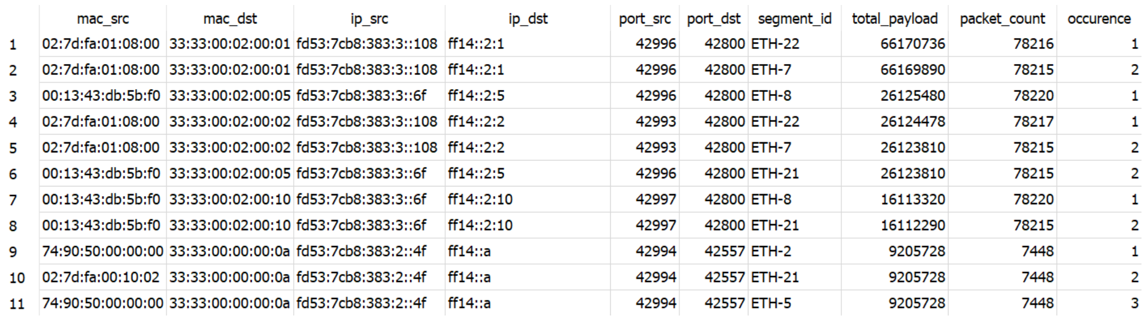
If such a trace is analyzed using common tools for PCAP analysis, such as Wireshark, or A-Packets, the resulting statistics (such as number of packets sent by each device), will be skewed in a way that some devices will seem to send more packets than is the reality. Therefore, to create a meaningful network statistics, it is necessary to filter out those packets.

With the way the data is stored in the database, the first solution that came to mind was to remove all the records that are the same in all fields except for the segment identifier (segment_id), which is unique for each measured segment. However, this approach works only for the packets that are captured on a single LAN or VLAN. When the packets are captured on multiple VLANs, the source and destination MAC is changed by the router, therefore, routed packets would not be filtered out. This approach is therefore used only for filtering non-IP records from the database that cannot be routed.

The way it works in the analyzer for IP records, which make up the majority of network traffic in a vehicle, is that the records in the database are partitioned by the source and destination IP address and the source and destination ports. Then each partition is ordered in descending order by payload. This ordering was selected because, especially for UDP communication, some packets may not get delivered. This means that the same stream can be captured multiple times, each time with a different payload and packet count. The ordering ensures that only the maximum payload observed for each stream is selected. Then records in every partition are assigned an index number as shown in the SQL query below. To get only unique records, all records from the FlowOccurences table are selected where the occurrence is equal to 1. The SQL query is built using a Python function that allows user to specify more filters, to get only records with specific addresses, ports, or protocols.

```
WITH FlowsOnSegment AS
  (SELECT mac_src,
          mac_dst,
          ip_src,
          ip_dst,
          port_src,
          port_dst,
          segment_id,
          SUM(payload) AS total_payload,
          SUM(cnt) AS packet_count
   FROM communication
   WHERE protocol LIKE "%ip%"
   GROUP BY ip_src,
            ip_dst,
            port_src,
              port_dst,
              segment_id),
       FlowOccurences AS
    (SELECT *,
            ROW_NUMBER() OVER (PARTITION BY ip_src,
                                            ip_dst,
                                            port_src,
                                            port_dst
                               ORDER BY total_payload DESC) AS ocurence
     FROM FlowsOnSegment)
  SELECT * FROM FlowOccurences
```

The sample of a result of running the query above can be seen in the picture 5.2. The records in rows 1 and 2 are obviously from the same flow, but in the second record there is one packet less, therefore only the first one is picked. The reason why MAC addresses cannot be used in partitioning of the records can be seen in rows 9 to 11, where all three records belong to the same flow, but the record in row 10 has the source MAC address of a router, as was detected in the network graph 4.8.



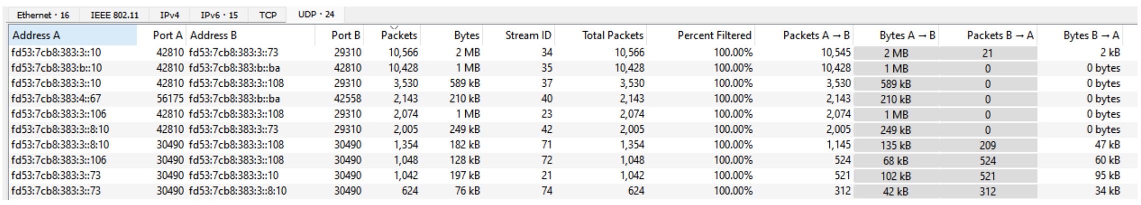
	mac_src	mac_dst	ip_src	ip_dst	port_src	port_dst	segment_id	total_payload	packet_count	occurence
1	02:7d:fa:01:08:00	33:33:00:02:00:01	fd53:7cb8:383:3::108	ff14::2:1	42996	42800	ETH-22	66170736	78216	1
2	02:7d:fa:01:08:00	33:33:00:02:00:01	fd53:7cb8:383:3::108	ff14::2:1	42996	42800	ETH-7	66169890	78215	2
3	00:13:43:db:5b:f0	33:33:00:02:00:05	fd53:7cb8:383:3::6f	ff14::2:5	42996	42800	ETH-8	26125480	78220	1
4	02:7d:fa:01:08:00	33:33:00:02:00:02	fd53:7cb8:383:3::108	ff14::2:2	42993	42800	ETH-22	26124478	78217	1
5	02:7d:fa:01:08:00	33:33:00:02:00:02	fd53:7cb8:383:3::108	ff14::2:2	42993	42800	ETH-7	26123810	78215	2
6	00:13:43:db:5b:f0	33:33:00:02:00:05	fd53:7cb8:383:3::6f	ff14::2:5	42996	42800	ETH-21	26123810	78215	2
7	00:13:43:db:5b:f0	33:33:00:02:00:10	fd53:7cb8:383:3::6f	ff14::2:10	42997	42800	ETH-8	16113320	78220	1
8	00:13:43:db:5b:f0	33:33:00:02:00:10	fd53:7cb8:383:3::6f	ff14::2:10	42997	42800	ETH-21	16112290	78215	2
9	74:90:50:00:00:00	33:33:00:00:00:0a	fd53:7cb8:383:2::4f	ff14::a	42994	42557	ETH-2	9205728	7448	1
10	02:7d:fa:00:10:02	33:33:00:00:00:0a	fd53:7cb8:383:2::4f	ff14::a	42994	42557	ETH-21	9205728	7448	2
11	74:90:50:00:00:00	33:33:00:00:00:0a	fd53:7cb8:383:2::4f	ff14::a	42994	42557	ETH-5	9205728	7448	3

Figure 5.2. Data returned by query.

The filtered data are used for displaying statistics and visualization of the flows.

5.2 Network statistics

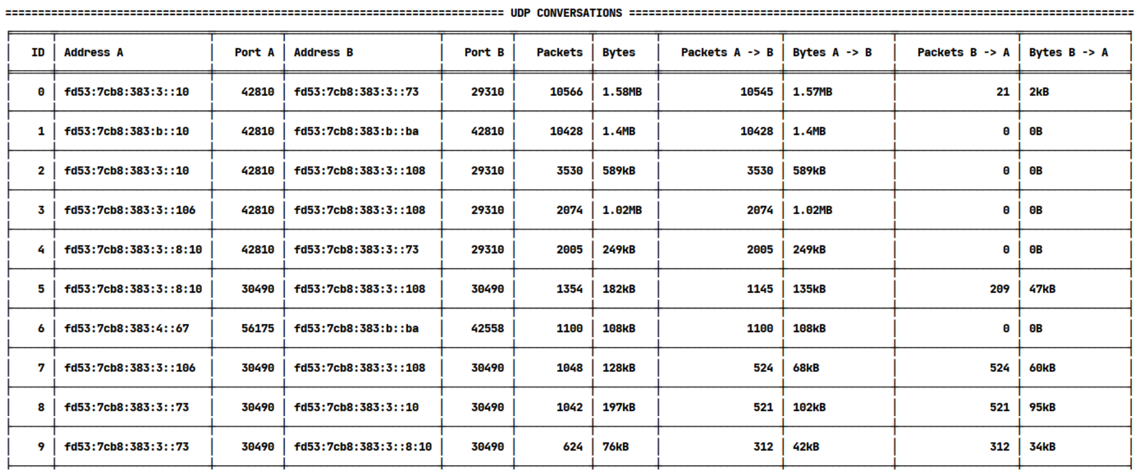
Currently, there are two statistic functions implemented, inspired by those available in Wireshark - endpoints and conversations. The difference between the statistics from the developed tool and the Wireshark statistics can be seen when the same log file that was used for network reconstruction, shown in Figure 4.8, is used for statistics. As an example, the figures below show the result of the top 10 unicast UDP conversations, Figure 5.3 from Wireshark, and Figure 5.4 from the developed tool. The fourth and fifth conversation from Wireshark got shifted to seventh and eight positions, and the number of sent packets and bytes is (almost, due to packet losses) half. This is no surprise, as when the path between these nodes is checked in Figure 4.8, it is clear that the packets traveled through two measured edges. In contrast, the first three conversations are the same, because there is only one measured edge in the path between the nodes.



Ethernet · 16 | IEEE 802.11 | IPv4 | IPv6 · 15 | TCP | UDP · 24

Address A	Port A	Address B	Port B	Packets	Bytes	Stream ID	Total Packets	Percent Filtered	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A
fd53:7cb8:383:3::10	42810	fd53:7cb8:383:3::73	29310	10,566	2 MB	34	10,566	100.00%	10,545	2 MB	21	2 kB
fd53:7cb8:383:b::10	42810	fd53:7cb8:383:b::ba	42810	10,428	1 MB	35	10,428	100.00%	10,428	1 MB	0	0 bytes
fd53:7cb8:383:3::10	42810	fd53:7cb8:383:3::108	29310	3,530	589 kB	37	3,530	100.00%	3,530	589 kB	0	0 bytes
fd53:7cb8:383:4::67	56175	fd53:7cb8:383:b::ba	42558	2,143	210 kB	40	2,143	100.00%	2,143	210 kB	0	0 bytes
fd53:7cb8:383:3::106	42810	fd53:7cb8:383:3::108	29310	2,074	1 MB	23	2,074	100.00%	2,074	1 MB	0	0 bytes
fd53:7cb8:383:3::8:10	42810	fd53:7cb8:383:3::73	29310	2,005	249 kB	42	2,005	100.00%	2,005	249 kB	0	0 bytes
fd53:7cb8:383:3::8:10	30490	fd53:7cb8:383:3::108	30490	1,354	182 kB	71	1,354	100.00%	1,145	135 kB	209	47 kB
fd53:7cb8:383:3::106	30490	fd53:7cb8:383:3::108	30490	1,048	128 kB	72	1,048	100.00%	524	68 kB	524	60 kB
fd53:7cb8:383:3::73	30490	fd53:7cb8:383:3::10	30490	1,042	197 kB	21	1,042	100.00%	521	102 kB	521	95 kB
fd53:7cb8:383:3::73	30490	fd53:7cb8:383:3::8:10	30490	624	76 kB	74	624	100.00%	312	42 kB	312	34 kB

Figure 5.3. Top 10 unicast UDP conversations returned by Wireshark.



== UDP CONVERSATIONS ==

ID	Address A	Port A	Address B	Port B	Packets	Bytes	Packets A -> B	Bytes A -> B	Packets B -> A	Bytes B -> A
0	fd53:7cb8:383:3::10	42810	fd53:7cb8:383:3::73	29310	10566	1.58MB	10545	1.57MB	21	2kB
1	fd53:7cb8:383:b::10	42810	fd53:7cb8:383:b::ba	42810	10428	1.4MB	10428	1.4MB	0	0B
2	fd53:7cb8:383:3::10	42810	fd53:7cb8:383:3::108	29310	3530	589kB	3530	589kB	0	0B
3	fd53:7cb8:383:3::106	42810	fd53:7cb8:383:3::108	29310	2074	1.02MB	2074	1.02MB	0	0B
4	fd53:7cb8:383:3::8:10	42810	fd53:7cb8:383:3::73	29310	2005	249kB	2005	249kB	0	0B
5	fd53:7cb8:383:3::8:10	30490	fd53:7cb8:383:3::108	30490	1354	182kB	1145	135kB	209	47kB
6	fd53:7cb8:383:4::67	56175	fd53:7cb8:383:b::ba	42558	1100	108kB	1100	108kB	0	0B
7	fd53:7cb8:383:3::106	30490	fd53:7cb8:383:3::108	30490	1048	128kB	524	68kB	524	60kB
8	fd53:7cb8:383:3::73	30490	fd53:7cb8:383:3::10	30490	1042	197kB	521	102kB	521	95kB
9	fd53:7cb8:383:3::73	30490	fd53:7cb8:383:3::8:10	30490	624	76kB	312	42kB	312	34kB

Figure 5.4. Top 10 unicast UDP conversations returned by developed tool.

5.3 Data-flow diagram

To visualize data-flows between devices, the topology graph that keeps the nodes separate for every IP address was used, such as the graph shown in Figure 4.8. The reason for using this graph instead of the graph representing the physical topology is mainly because it is easier to find the path of the packet when the devices from different VLAN communicate. If the connection between the switch and router that routes the packets between VLANs is represented by a trunk port, the shortest path between the nodes in the graph is only through a switch, but the path from switch to router and back from router to switch is missing.

If a data-flow diagram represented in a physical topology graph is needed, such as the topology graph shown in Figure 4.9, the data-flow diagram can be converted to physical by merging the switches connected to the same router interface and merging the nodes with the same MAC address (and adding data-flows from corresponding edges), or another option would be to develop a more complicated pathfinding algorithm that would take routing into account.

To create a data-flow diagram, the topology graph is first converted to a directed graph. Then the data-flows are extracted from the database using the query from Section 5.1, which allows user to specify protocol, source and destination address, source and destination port and whether to use only unicast packets.

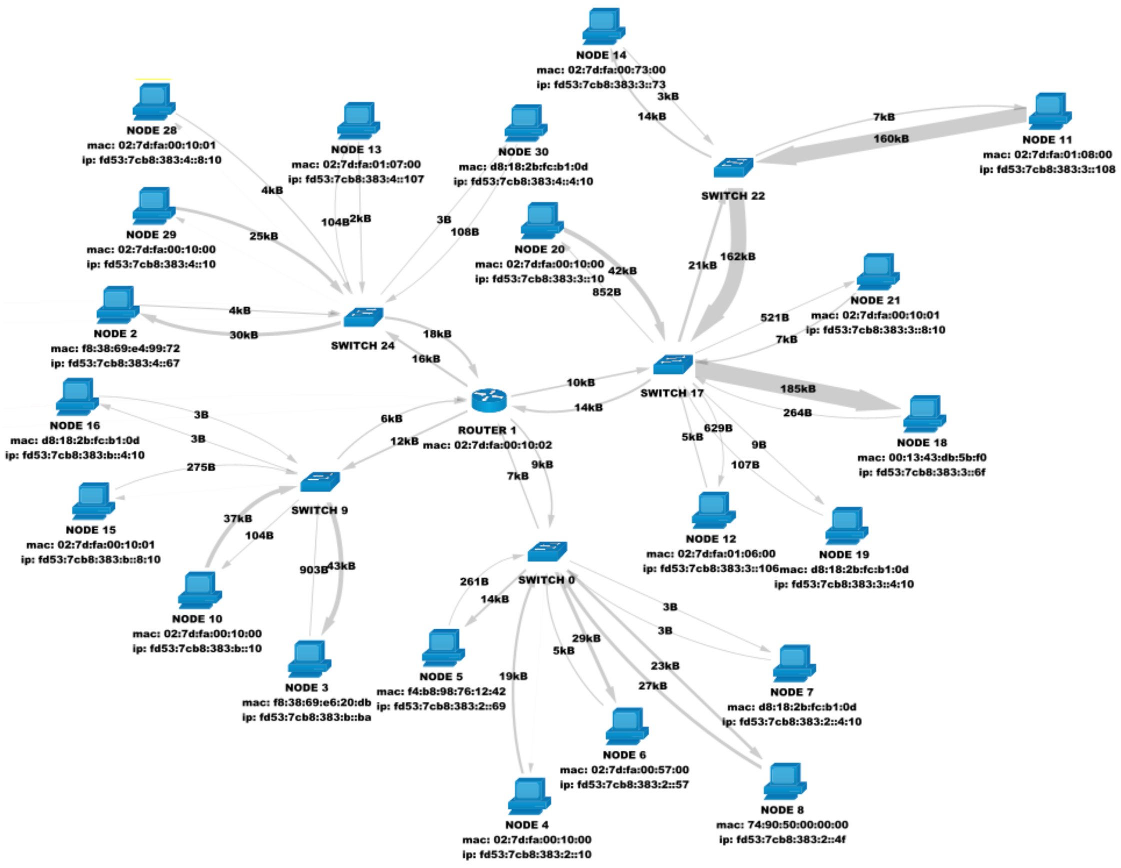


Figure 5.5. UDP dataflow diagram

Then, for each data-flow, the endpoints are localized in the graph (if IP addresses are available by IP, otherwise by MAC), and the shortest path between them is found using the built-in igraph method. If the flow is unicast, for each edge in the path the payload of the flow is added to it. For multicast flows there are multiple paths to multiple endpoints; therefore, for each edge also the source and destination port is kept, and the payload is added to the edge only if the ports are not present, ensuring that the same data-flow is not present in one edge multiple times. Membership in multicast groups is determined only for the nodes that were on the ends of the measured edge, as for these nodes, it can be reliably determined to which multicast groups they subscribe.

In the end, the width of the edges is normalized according to the payload, resulting in a data-flow diagram as shown in Figure 5.5

Chapter 6
Online analysis

This chapter explains the possibilities of getting more information about the network by sending packets from the TAP device. The TAP device used is Vector VN 5620 with CANoe software. There are two basic ways how to send the packets, one is directly from CANoe, using the built-in Ethernet Packet Generator, or by connecting a PC to the VN device configured as a switch and sending the packets from computer using other software, for example python Scapy library. For purposes of the thesis, only Ethernet Packet Generator was used, as the goal was to test whether it is possible to get some useful information from the network, and this tool is sufficient (although not as flexible). One thing to note is that when connecting TAP to the network, it is a good idea to capture outgoing packets from one of the devices on the network to get its IP and MAC address and use these addresses as the source addresses for the packets, since the car network usually has some network access restrictions.

6.1 Detecting routers using the TTL field

Looking back at the reconstructed topology of the simulated network shown in Figure 4.7, there was an issue that the algorithm could not decide, whether the two detected router vertices are two interfaces of the same router, or whether there are multiple routers in the network.

To distinguish these two situations, the TTL field (hop limit in IPv6) can be used. The TLL field is specified for each IP packet and every time the packet passes through a router, its value decreases by one. If the TTL value of the packet reaches zero before reaching the destination, the router discards the packet and sends an ICMP message back to the sender.

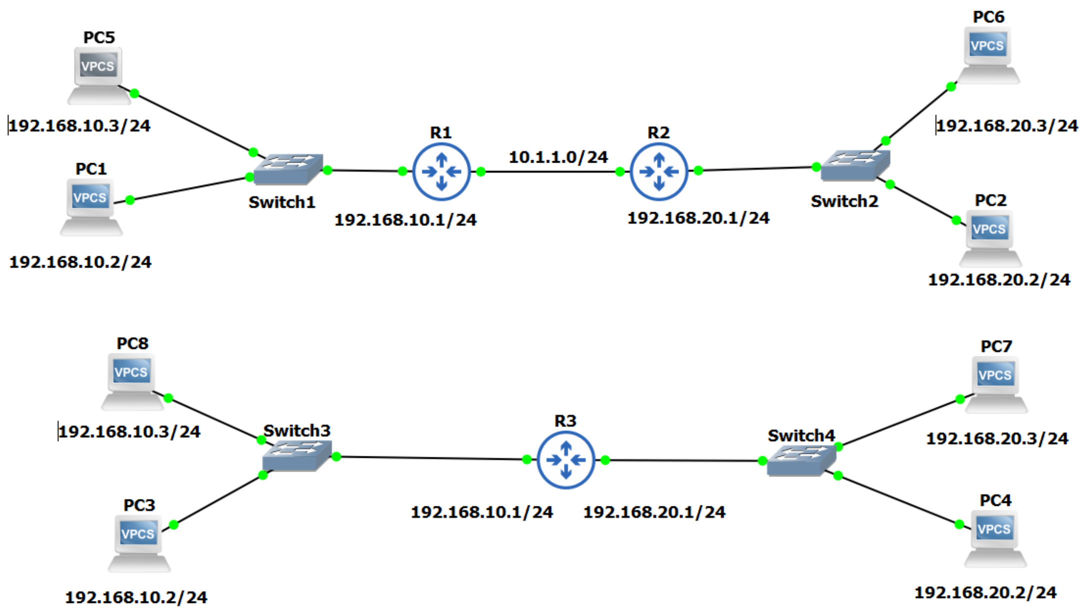


Figure 6.1. Network example in GNS3

Figure 6.1 shows an example of two very simple networks. In this example, consider that all end devices are communicating together and packets are captured at the links between switches and routers. The topology returned by the algorithm will be the same for both networks, as seen in Figure 6.2.

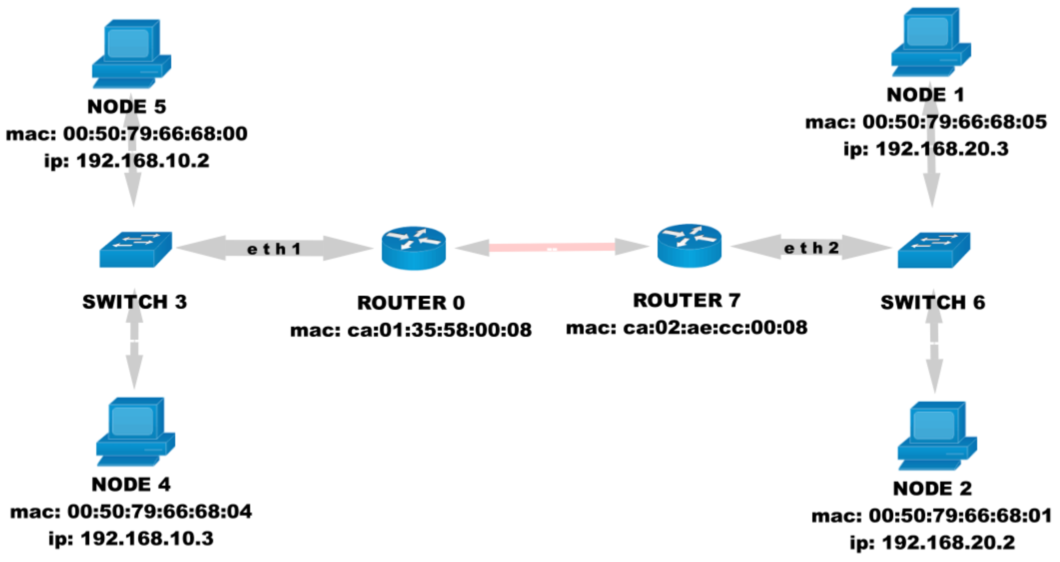
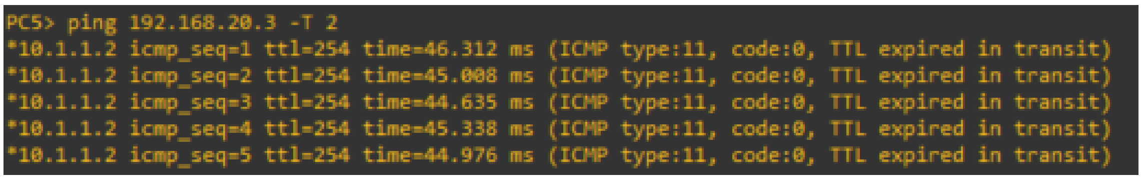


Figure 6.2. Reconstructed network topology

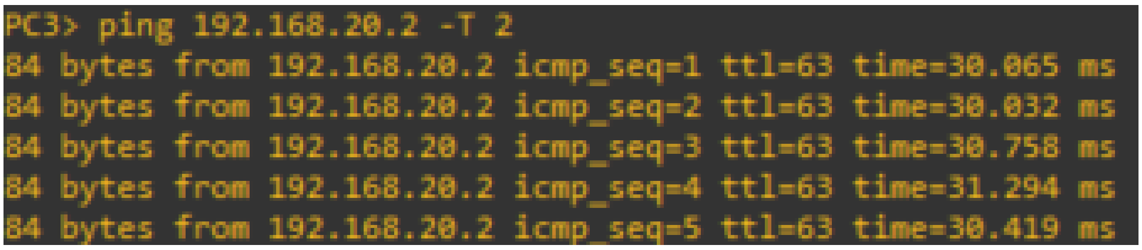
To distinguish between the two cases, the user can ping the device behind the router, first with the TTL set to 2, and continue increasing the TTL value until the device responds - in a similar way that the traceroute command works. Keep in mind that the router will first decrease the TTL value, and if it is equal to zero, it will drop the packet. Therefore, the number of routers in the path is equal to TTL - 1 for the lowest TTL that allows the packet to reach destination. If a packet with TTL equal to 2 is sent from PC1 to PC6 in Figure 6.1, PC1 will receive an ICMP response type 11 code 0 (TTL expired in transmit). If the TTL is increased to a value of three, the PC6 will respond to the ping, therefore there are two routers between PC1 and PC6.



```
PC5> ping 192.168.20.3 -T 2
*10.1.1.2 icmp_seq=1 ttl=254 time=46.312 ms (ICMP type:11, code:0, TTL expired in transit)
*10.1.1.2 icmp_seq=2 ttl=254 time=45.008 ms (ICMP type:11, code:0, TTL expired in transit)
*10.1.1.2 icmp_seq=3 ttl=254 time=44.635 ms (ICMP type:11, code:0, TTL expired in transit)
*10.1.1.2 icmp_seq=4 ttl=254 time=45.338 ms (ICMP type:11, code:0, TTL expired in transit)
*10.1.1.2 icmp_seq=5 ttl=254 time=44.976 ms (ICMP type:11, code:0, TTL expired in transit)
```

Figure 6.3. Ping response with TTL equal to 2, sent from PC1 to PC6.

Similarly, if a packet is sent from PC3 to PC4 with a TTL value of 2, the PC4 will respond, and it is clear that there is only one router in the path.



```
PC3> ping 192.168.20.2 -T 2
84 bytes from 192.168.20.2 icmp_seq=1 ttl=63 time=30.065 ms
84 bytes from 192.168.20.2 icmp_seq=2 ttl=63 time=30.032 ms
84 bytes from 192.168.20.2 icmp_seq=3 ttl=63 time=30.758 ms
84 bytes from 192.168.20.2 icmp_seq=4 ttl=63 time=31.294 ms
84 bytes from 192.168.20.2 icmp_seq=5 ttl=63 time=30.419 ms
```

Figure 6.4. Ping response with TTL equal to 2, sent from PC3 to PC4.

6.2 Testing on the HIL

To check the possibilities of using DoIP for network analysis, the HIL built from Skoda Enyaq units was used. More information on HIL (schematics and use) can be found in [7]. HIL is available in the Department of Measurement laboratory and consists of 6 units [7]:

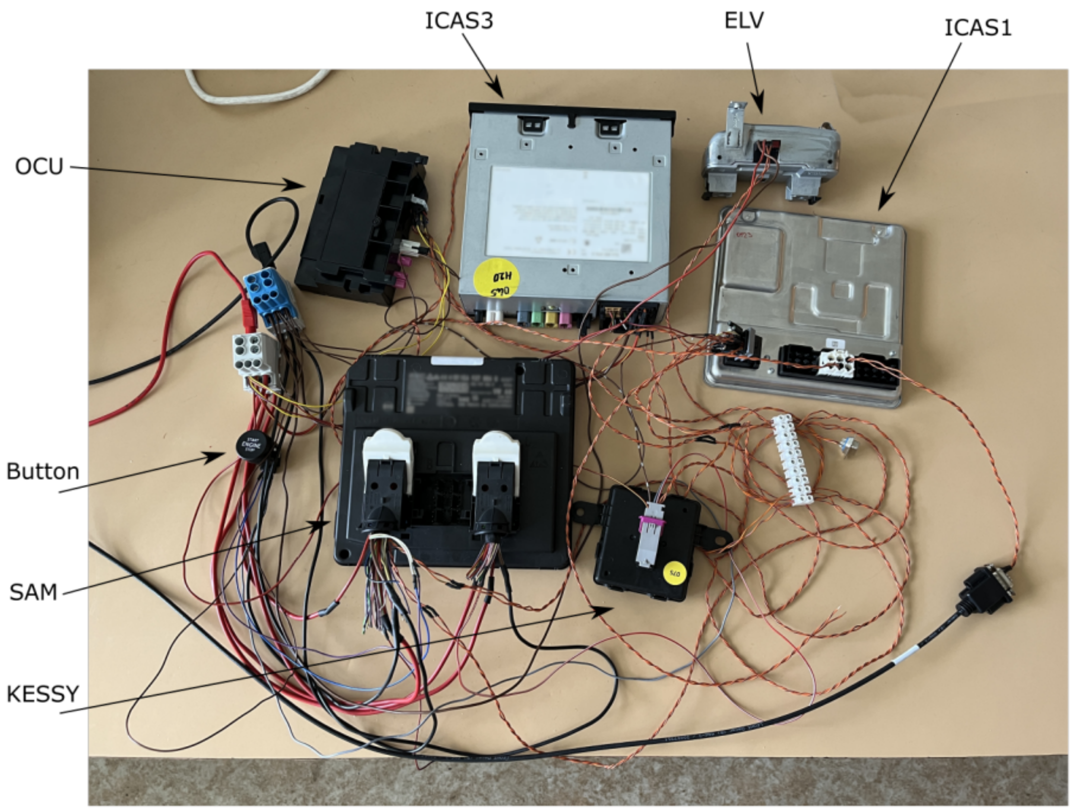
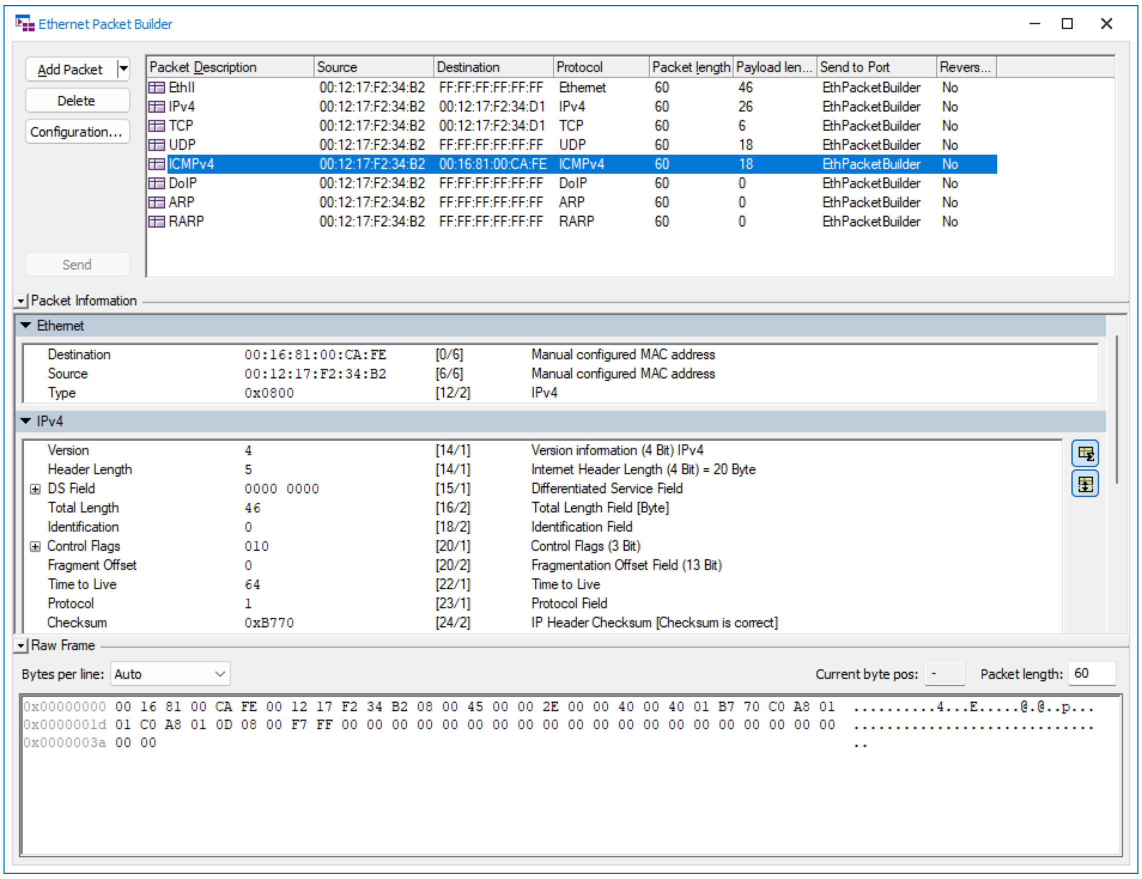


Figure 6.5. Photo of the connected HIL [7].

- In Car Application Server (ICAS) 1 - server responsible for functionality of driving assistance systems, AC, airbags, locking
- In Car Application Server (ICAS) 3 - server for digital devices, such as infotainment or head-up display
- Online Connectivity Unit (OCU) - telematics device, reponsible for wireless communications
- Signal Acquisition Module (SAM) - unit for receiving data from sensors and controllers
- Electronic Steering Lock (ELV) - unit reponsible for blocking the steering wheel
- Keyless Entry Start and Exit System (KESSY) - unit for keyless access and starting
- Button

The connection between ICAS 1 and ICAS 3 was split and connected through a Vector VN 5620 device to a monitoring computer with CANoe 15 installed. The VN 5620 was configured as a switch, with ICAS 1 connected on one port, ICAS 3 connected to a second port, and on the third port was an Ethernet Packet Builder (Figure 6.6), which allows one to build and send the packets to the network.



Packet Description	Source	Destination	Protocol	Packet length	Payload len...	Send to Port	Revers...
EthII	00:12:17:F2:34:B2	FF:FF:FF:FF:FF:FF	Ethernet	60	46	EthPacketBuilder	No
IPv4	00:12:17:F2:34:B2	00:12:17:F2:34:D1	IPv4	60	26	EthPacketBuilder	No
TCP	00:12:17:F2:34:B2	00:12:17:F2:34:D1	TCP	60	6	EthPacketBuilder	No
UDP	00:12:17:F2:34:B2	FF:FF:FF:FF:FF:FF	UDP	60	18	EthPacketBuilder	No
ICMPv4	00:12:17:F2:34:B2	00:16:81:00:CA:FE	ICMPv4	60	18	EthPacketBuilder	No
DoIP	00:12:17:F2:34:B2	FF:FF:FF:FF:FF:FF	DoIP	60	0	EthPacketBuilder	No
ARP	00:12:17:F2:34:B2	FF:FF:FF:FF:FF:FF	ARP	60	0	EthPacketBuilder	No
RARP	00:12:17:F2:34:B2	FF:FF:FF:FF:FF:FF	RARP	60	0	EthPacketBuilder	No

Ethernet

Destination	00:16:81:00:CA:FE	[0/6]	Manual configured MAC address
Source	00:12:17:F2:34:B2	[6/6]	Manual configured MAC address
Type	0x0800	[12/2]	IPv4

IPv4

Version	4	[14/1]	Version information (4 Bit) IPv4
Header Length	5	[14/1]	Internet Header Length (4 Bit) = 20 Byte
⊞ DS Field	0000 0000	[15/1]	Differentiated Service Field
Total Length	46	[16/2]	Total Length Field [Byte]
Identification	0	[18/2]	Identification Field
⊞ Control Flags	010	[20/1]	Control Flags (3 Bit)
Fragment Offset	0	[20/2]	Fragmentation Offset Field (13 Bit)
Time to Live	64	[22/1]	Time to Live
Protocol	1	[23/1]	Protocol Field
Checksum	0xB770	[24/2]	IP Header Checksum [Checksum is correct]

Bytes per line: Auto   Current byte pos: -   Packet length: 60

```
0x00000000 00 16 81 00 CA FE 00 12 17 F2 34 B2 08 00 45 00 00 2E 00 00 40 00 40 01 B7 70 C0 A8 01  ..........4...E.....@.@..p...
0x0000001d 01 C0 A8 01 0D 08 00 F7 FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .............................
0x0000003a 00 00                                                                                     ..
```

Figure 6.6. Ethernet packet builder.

From the ICMPv6 protocol, an idea was to remove the limitation that only devices that communicate on the network are visible to the algorithm. This can be solved by pinging all the end devices on the LAN, using the ICMPv6 ping by sending an Echo Request message to a multicast address `FF02::1`. This has to be done on every VLAN in the network, as the scope of the address is only link-local. Similarly, ping sent to a `FF02::2` address is sent to all routers to the network, which will give us the missing IP address of the router, which the algorithm cannot detect.

Another idea was to use the Neighbor Discovery Protocol (NDP) to get the MAC addresses of the unknown devices, but it is not really useful for this purpose, because it works only on the local network. If the tester is already connected to the LAN with the unknown device, the packets captured from the multicast ping will already make the MAC address of the unknown node visible to the algorithm.

From the DoIP protocol, an use-case for improving the function of the algorithm was not found. Mainly because everything except vehicle announcement requires diagnostic address of the ECUs that is not known, and vehicle announcement does not provide any additional information about the network. One thing to note is that at the start of the vehicle, every DoIP enabled

ECU sends a vehicle announcement message to a DoIP gateway, therefore capturing the packets during the startup of the vehicle may improve number of detected devices in the network.

Chapter 7
Conclusion

The first part of the thesis described how Ethernet is used in vehicles and how such networks can be monitored. In the implementation part, the capture file was first parsed into a database to make access to relevant data quicker. Based on the fact that TAP devices allow one to separate the network traffic by direction and that the addresses of the devices in the automotive network are static for security reasons, an algorithm to create topology graphs from every measured segment was proposed and developed. Another algorithm merged multiple segment graphs into an overall network topology. The topology can be further exported to a graphml file and analyzed in various graph visualization tools, such as Gephi or Cytoscape.

The topology graphs and the data from the database were further used to visualize the dataflows in the network, allowing user to specify requested devices, protocols or ports. During the work I realized that using the Multi-TAP device for network monitoring causes that some packets are captured multiple times, resulitng in skewed statistics. A procedure to get rid of the duplicate records was presented. Then the functions to present corrected statistics were implemented, as to filter out duplicates from the original log file would be more complicated.

To overcome lack of real-world data, during the development a GNS3 network simulator was used to simulate various network configurations, however it was also tested on data measured on an Ethernet part of the network of Skoda Enyaq vehicle.

In the end, some options were explored to improve the results of the analysis using active polling. Some limitations were identified during the testing, for example, that the current implementation of the algorithm cannot identify whether the two interfaces belong to the same router or not, which can be solved by sending packets with increasing TTL. Routers and devices can be forced to answer using multicast ping. The plan was to also use Network Discovery Protocol, but in the end it showed that it does not provide any useful information compared to ping for the current use case.

The tool can be easily expanded for more functionality - the data from the logs is exported using tshark, therefore all packet fields available in Wireshark may be accessed by the tool by adding the fields in database controller. For each detected device the information about it can be stored in its attributes and the visualization can be configured to visualize these attributes as needed.

References

[1] K. Matheus, and T. Königseder. *Automotive Ethernet*. Cambridge University Press, 2021. ISBN 9781108841955.
`https://books.google.cz/books?id=IjMiEAAAQBAJ`.

[2] C.M. Kozierok. *Automotive Ethernet: The Definitive Guide ; [TCP/IP, BroadR-Reach, Switch Technology, Real-Time Protocols, Audio Video Bridging, IEEE Physical Layers, Electromagnetic Compatibility & More*. Intrepid Control Systems, 2014. ISBN 9780990538806.
`https://books.google.cz/books?id=GlqhoAEACAAJ`.

[3] *About OPEN Alliance*. 2024.
`https://opensig.org/about/`.

[4] D. Paret, H. Rebaine, and B.A. Engel. *Autonomous and Connected Vehicles: Network Architectures from Legacy Networks to Automotive Ethernet*. Wiley, 2022. ISBN 9781119816133.
`https://books.google.cz/books?id=6ZtkEAAAQBAJ`.

[5] C.E. Spurgeon, and J. Zimmerman. *Ethernet: The Definitive Guide*. O'Reilly, 2014. ISBN 9781449361846.
`https://books.google.cz/books?id=UcXanQEACAAJ`.

[6] G. Lee. *Cloud Networking: Understanding Cloud-based Data Center Networks*. Elsevier Science, 2014. ISBN 9780128008164.
`https://books.google.cz/books?id=7QVOAwAAQBAJ`.

[7] Jakub Hortenský. *SOME/IP Implementation for Testing*. 2023.

[8] Muzhir Al-Ani, and Rola A.A.Haddad. IPv4/IPv6 Transition. *International Journal of Engineering Science and Technology*. 2012, 4 4815-4822.

[9] A.S. Tanenbaum, and D.J. Wetherall. *Computer Networks*. Pearson Education, 2012. ISBN 9780133072624.
`https://books.google.cz/books?id=IRUvAAAAQBAJ`.

[10] C.E. Spurgeon, and J. Zimmerman. *Ethernet Switches*. O'Reilly Media, 2013. ISBN 9781449367305.
`https://books.google.cz/books?id=_-ak0iJuHqQC`.

[11] Kai Jansen. *Getting started with Ethernet VN Devices*. 5.10.2021.
`https://cdn.vector.com/cms/content/know-how/_application-`

notes/AN-ANI-1-116_Getting_started_with_Ethernet_Interfaces.pdf.

[12] *ping*. 2023.
https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/tracert.

[13] *tracert*. 2023.
https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/tracert.

[14] *Wireshark: A Network Protocol Analyzer*. 2024.
https://www.wireshark.org/.

[15] *NetworkMiner - The NSM and Network Forensics Analysis Tool*. 2024.
https://www.netresec.com/?page=NetworkMiner.

[16] *CANoe.Ethernet: Development and Test Tool CANoe.Ethernet*. 2024.
https://www.vector.com/in/en/products/products-a-z/software/canoe/option-ethernet/#.

[17] *NetCapVis: Analyse your local network traffic files*. 2024.
https://netcapvis.igd.fraunhofer.de/.

[18] *A-Packets: Unleash the Power of PCAP Analysis*. 2024.
https://apackets.com/.

[19] *About DynamiteLab*. 2024.
https://lab.dynamite.ai/about.

[20] *GRASSMARLIN User Guide*. 2024.
https://github.com/nsacyber/GRASSMARLIN/blob/master/GRASSMARLIN%20User%20Guide.pdf.

[21] *Getting Started with GNS3*. 2024.
https://docs.gns3.com/docs/.

[22] *TraceWrangler - Packet Capture Toolkit*. 2020.
https://www.tracewrangler.com/.

Appendix A
Segment graphs from the vehicle log

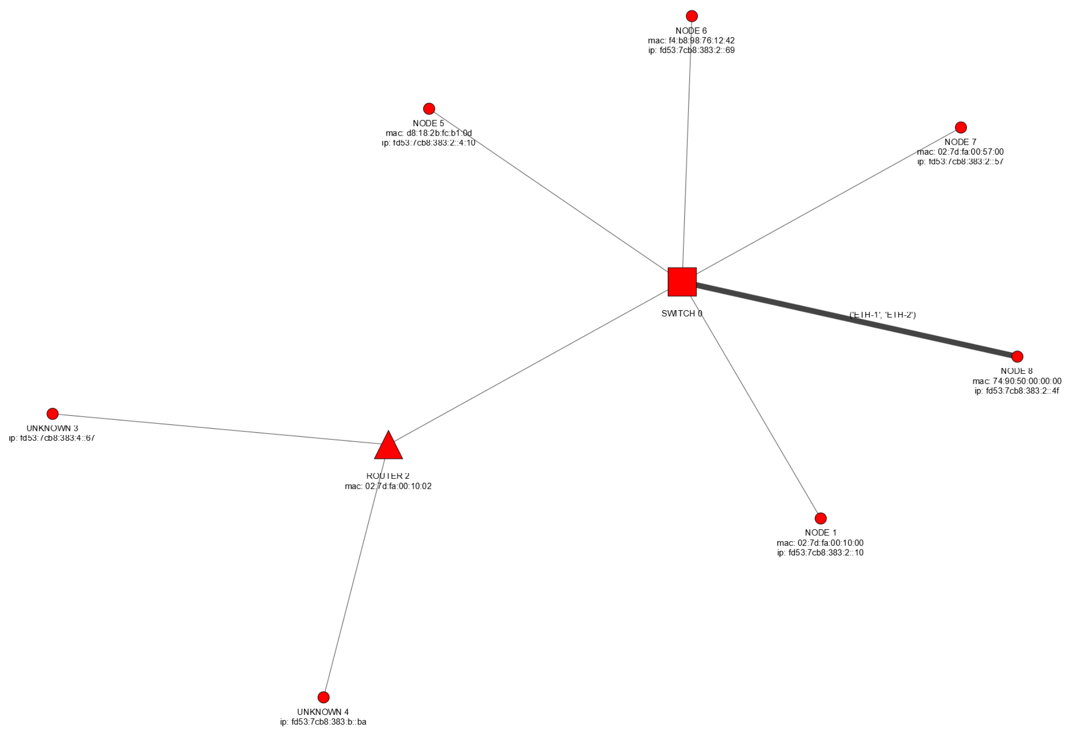


Figure A.1. Segment graph from segment ETH1-ETH2.

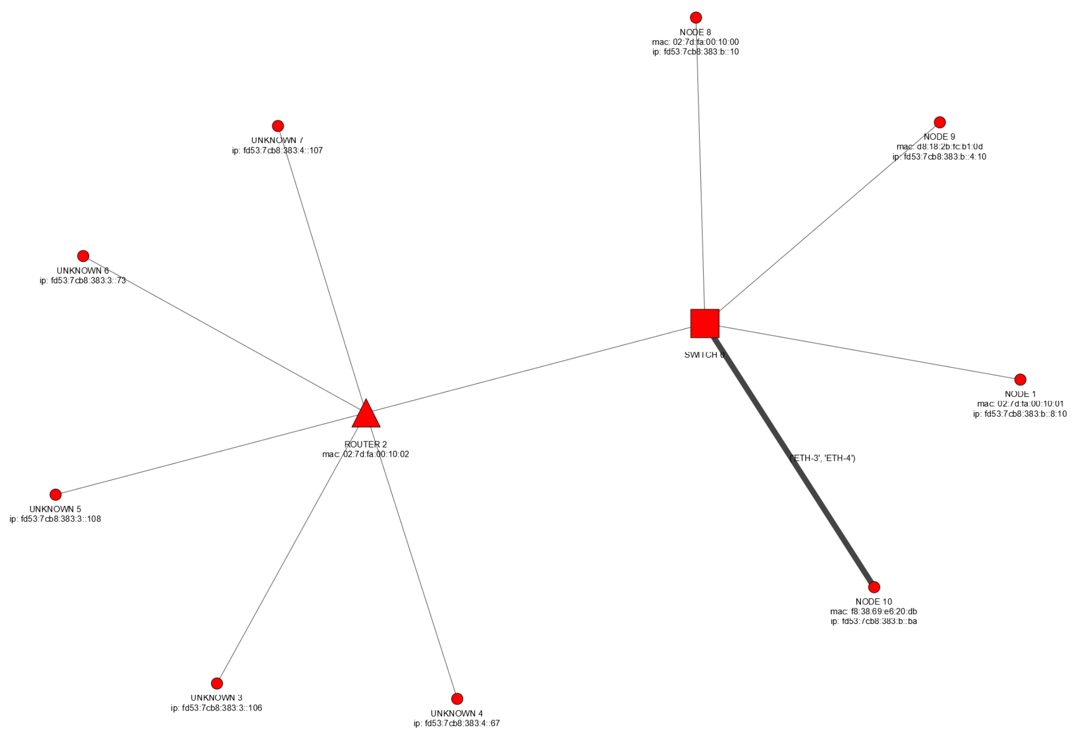


Figure A.2. Segment graph from segment ETH3-ETH4.

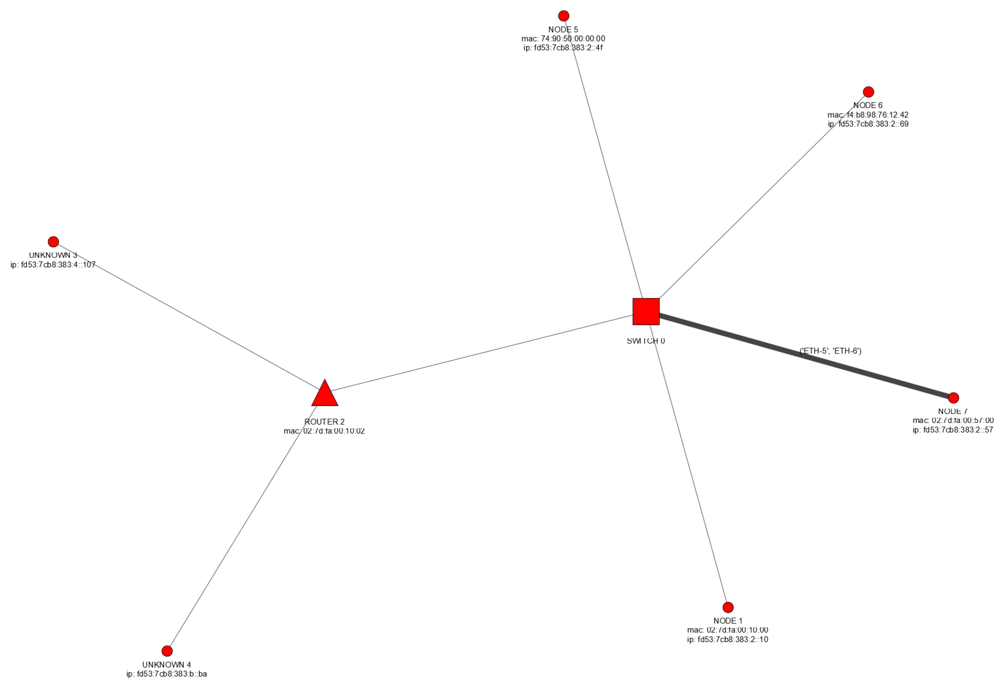


Figure A.3. Segment graph from segment ETH5-ETH6.

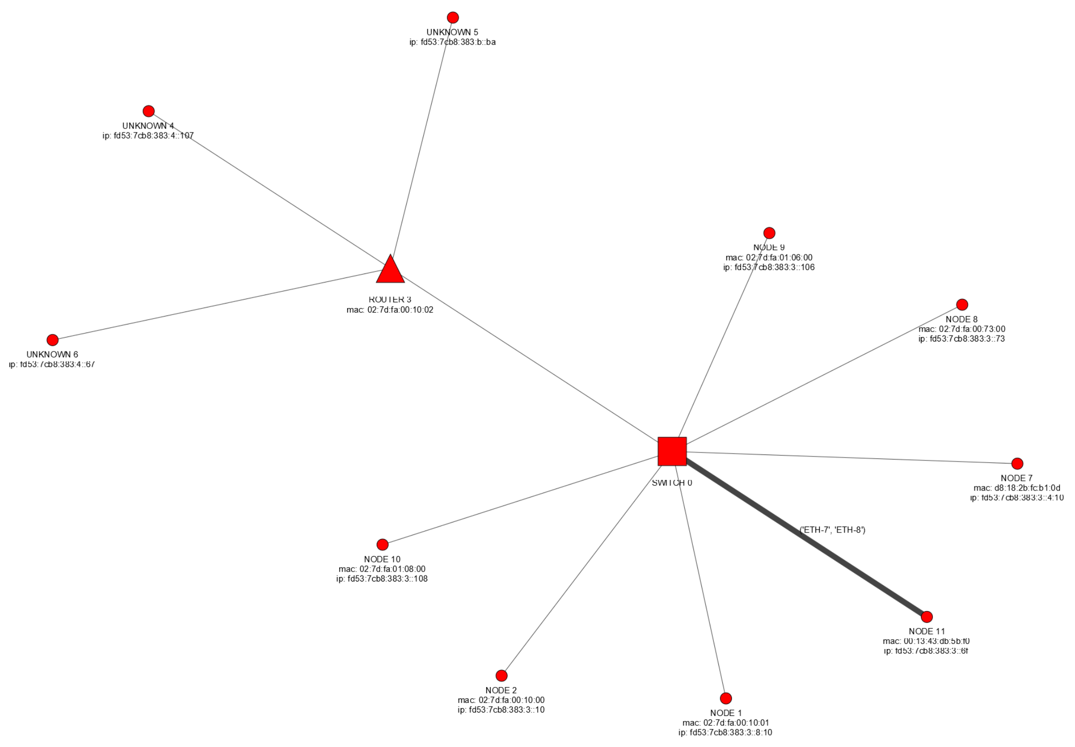


Figure A.4. Segment graph from segment ETH7-ETH8.

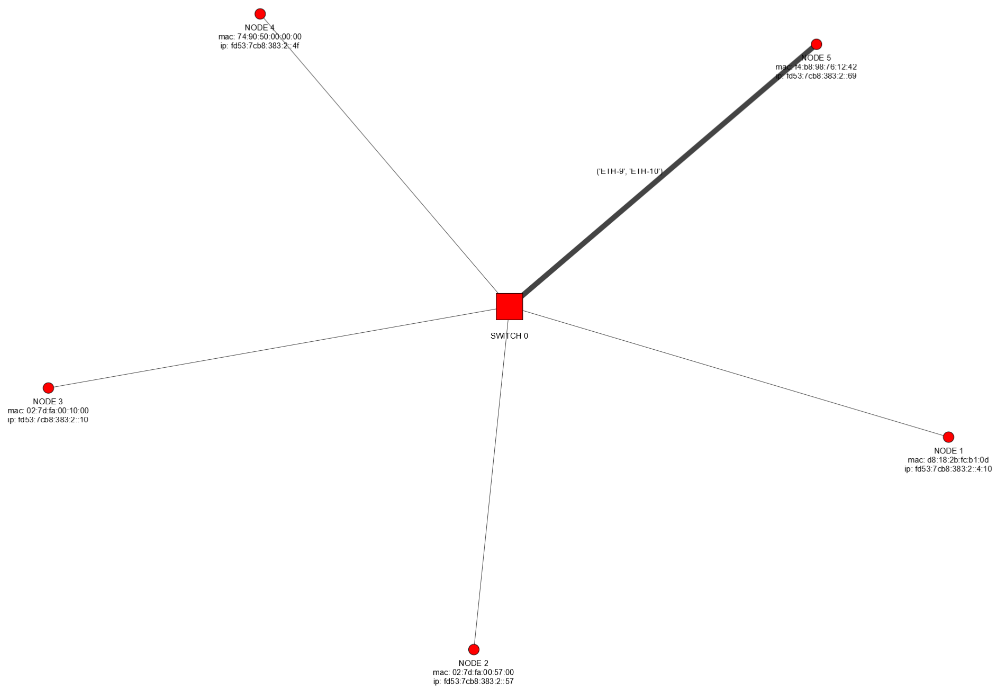


Figure A.5. Segment graph from segment ETH9-ETH10.

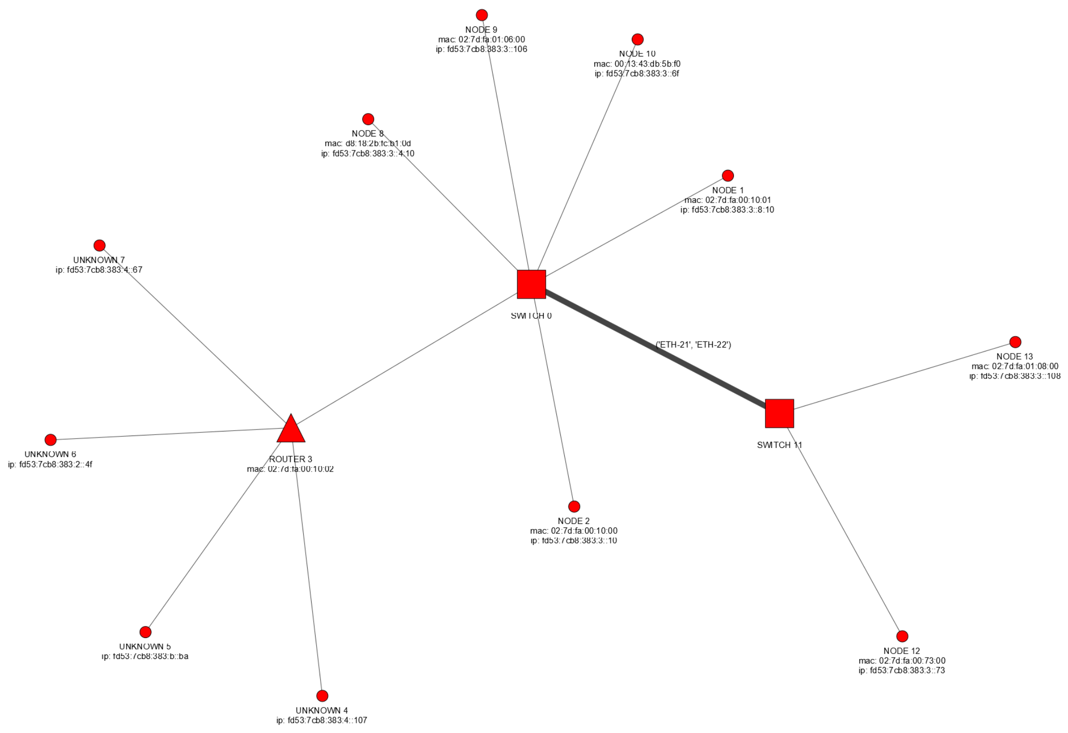


Figure A.6. Segment graph from segment ETH21-ETH22.

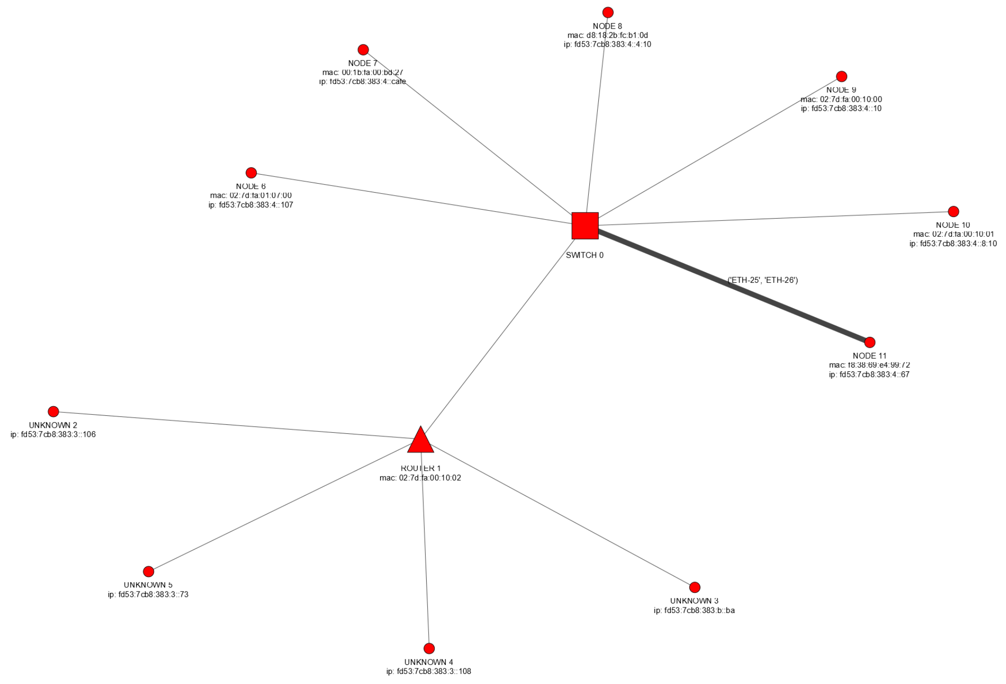


Figure A.7. Segment graph from segment ETH25-ETH26.

Appendix B
List of Abbreviations

ADAS	Advanced Assistance Systems
ASIC	Application Specific Integrated Circuit
CAN	Controller Area Network
DEI	Drop Eligible Indicator
DHCP	Dynamic Host Configuration Protocol
DoIP	Diagnostics Over Internet Protocol
ECU	Electronic Control Unit
ELV	Electronic Steering Lock
EMC	Electromagnetic Compatibility
GNS3	Graphical Network Simulator-3
HIL	Hardware In the Loop
ICAS	In Car Application Server
ICMP	Internet Control Message Protocol
ICMPv6	Internet Control Message Protocol Version 6
ID	Identifier
IEEE	Institute of Electrical and Electronics Engineers
IHL	IP Header Length
IOS	Internetworking Operating System
IP	Internet Protocol
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
KESSY	Keyless Entry Start and Exit System
L2	Layer 2
L3	Layer 3
LAN	Local Area Network
LIN	Local Interconnect Network
LiDAR	Light Detection And Ranging
MAC	Media Access Control
MOST	Media Oriented Systems Transport
Mbps	Megabits per second
NDP	Neighbor Discovery Protocol
OABR	OPEN Alliance BroadR-Reach

OCU	Online Connectivity Unit
OPEN	One-Pair Ether-Net
OUI	Organizationally Unique Identifier
PC	Personal Computer
PHY	Physical Layer
PIX	Private Internet eXchange
RAM	Random Access Memory
SAM	Signal Acquisition Module
SQL	Structured Query Language
TAP	Test Access Point
TCI	Tag Control Information
TCP	Transmission Control Protocol
TOS	Type of Service
TPID	Tag Protocol Identifier
TTL	Time To Live
UDP	User Datagram Protocol
UTP	Unshielded Twisted Pair
VID	VLAN Identifier
VLAN	Virtual Local Area Network