TECHNISCHE UNIVERSITÄT
CHEMNITZ

Realization of communication with SOME/IP stack over Ethernet

Research Internship

Submitted in Fulfilment of the
Requirements for the Academic Degree
M.Sc.

Dept. of Computer Science
Chair of Computer Engineering

Submitted by: Sreedhar Hegde
Student ID: 671453
Submitted On: 12.04.2022

Supervising tutor: Prof. Dr. W. Hardt

Company Supervising tutor: M.Eng. Rick Podszuweit

Company Overview

Acknowledgments

This report is written to fulfill the requirement of the academic degree of M.Sc. Automotive Software Engineering from Faculty of Computer Science, Department of Computer Engineering at the Technical University of Chemnitz. The internship work would not exist without the cooperation between TU Chemnitz and IAV GmbH.

First of all, I would like to express my gratitude to Prof. Dr. Wolfram Hardt for giving me the opportunity to work under his department. Also, I am grateful for his guidance and advice during my studies at TU Chemnitz.

I am grateful to my supervisor Mr. Rick Podszuweit and my manager Mr. Stephan Reichelt, IAV GmbH, for providing me with all facilities, tools and information to complete this work and to achieve the best results during all the phases of the work. Also, I would like to thank them for their guidance without which it would not be possible to successfully complete this internship.

Finally, I would like to express my gratitude to my family and friends for supporting me during my all education phases and encourage me to continue my studies in Germany.

Abstract

Contents

List of Figures

List of Tables

List of Abbreviations

ADAS	Advanced Driver Assistance Systems
ARXML	AUTOSAR XML
API	Application Program Interface
AUTOSAR	Automotive Open System Architecture
AVB	Audio Video Bridging
BSW	Basic Software
CAN	Controller Area Network
COVESA	Connected Vehicle Systems Alliance
DDS	Data Distribution Service
DoIP	Diognastic over IP
ECU	Electronic Control Unit
eSOC	Embedded Service-Oriented Communication
Eth	Ethernet
GUI	Graphical User Interface
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IP	Internet Protocol
ISO	International Organization for Standardization
IVN	In-vehicle Networking
JSON	JavaScript Object Notation
LIN	Local Interconnect Network
MAC	Media Access Control
MOST	Media Oriented Systems Transport
OEM	Original Equipment Manufacturer
OS	Operating Systems
OSI	Open System Interconnection
PDU	Protocol Data Unit
PHY	Physical Layer
POSIX	Portable Operating System Interface
RPC	Remote Procedure Calling
SOA	Service Oriented Architecture
SOME/IP	Scalable service-Oriented MiddlewarE over IP
SOME/IP-SD	Scalable service-Oriented MiddlewarE over IP- Service Discovery
SWC	Software Component
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
V2X	Vehicle-to-Anything communication

1 Introduction

Communication is essential in modern vehicles to establish a link between the ECUs in the network. In addition, as the number of ECUs and high-performance controllers grows, so does the need for more bandwidth than traditional in-vehicle networks such as CAN, Flexray, and MOST can provide. With the introduction of Ethernet into the automotive domain, bandwidths of up to 1 Gb/s can now be achieved within the vehicle network. The use of Ethernet benefits systems such as ADAS and infotainment significantly. However, in order to transmit and receive data at a significantly high data rate, a robust communication control mechanism is required. The use of Ethernet benefits systems such as ADAS and infotainment substantially. However, in order to transmit and receive data at a remarkably high data rate, a robust communication control mechanism is required. With the growing interest in POSIX-based systems in the automotive domain, service oriented architecture (SOA) plays an important role in meeting the needs of technology-driven applications. The core of SOA is remote procedure calling (RPC) and the Client-Server mechanism. To realize these concepts, there is a need for a middleware that is specifically designed to run automotive applications smoothly. To accomplish this, SOME/IP middleware was introduced in the automotive context. As more applications migrate to Adaptive AUTOSAR, SOME/IP is well suited to serve as a communication control protocol alongside existing communication technologies.

In this report, a detailed study of the SOME/IP technology is conducted. In order to understand the working of SOME/IP technology, the open source library vsomeip offered by GENIVI is used. A demonstrator consisting of target hardware running with different underlying architectures such as x64, armv7 and armv8 is setup. The devices are connected with each other on a network using Ethernet. The working is realized by running applications based on vsomeip stack on theses hardware devices. Also, a troubleshooting guide consisting of the commonly faced issues and faults while using the SOME/IP technology have been documented as a reference document.

2 Literature Survey

2.1 Communication technologies in Automotive Domain

More than 100 ECUs connect via in-vehicle buses in modern vehicles[2]. As a result, there is a greater need than ever for a dependable communication network with high bandwidth. In order to meet these requirements, BMW implemented Ethernet for the first time in vehicles in 2013. At the same time, it is important to note that the incorporation of Ethernet as an in-vehicle networking system does not imply that traditional communication networks such as CAN, LIN, and MOST are rendered obsolete. Because these networks are robust, inexpensive, time-tested, and provide necessary performance for many applications, Automotive Ethernet will not completely replace them, but will supplement them to provide even more cost, performance, and feature benefits. Table 2.1 shows the important characteristics of automotive networks in comparison with the Ethernet.

Property	**Ethernet**	**CAN**	**FlexRay**	**MOST**	**LIN**
Bandwidth(Mb/s)	>100	1	20	150	0.02
Nodes	Scalable	30	22	64	16
Network Length	15m per link	40m	24m	1280m	40m
Topologies	Star, Tree	Bus	Bus,start	Ring,Star	Bus
Cost	High	Low	Low	High	Very low
Cabling	UTP	UTP	UTP	Optical,UTP	1-wire

Table 2.1: Comparison of important characteristics of automotive networks [1, p.202]

2.2 Ethernet in the Automotive Domain

In the year 1980, consumer-oriented Ethernet was introduced. However, Ethernet use in the automotive domain did not begin until 2013. This was due to EMC emissions levels being higher than those required for vehicle use. The introduction of BroadR-Reach twisted pair cables meant that the stringent EMC performance regulations were met and that the cables could finally be used in the automotive domain. Initially, Ethernet 100BASE-TX was used for OBD and updating the ECUs' flash memories[3]. It has also been used for applications relating to infotainment and camera systems over the years. With an increasing number of sensors in a vehicle, data acquisition and high-speed communication become essential. In the current scenario, Automotive Ethernet appears to be the most viable solution for meeting these requirements. Also, Automotive Ethernet is said to be the next-generation in-vehicle networking systems when connecting application domains, transporting different kinds of data (control data, streaming, etc.)[3].

2.3 Ethernet as backbone in vehicles

Traditionally, CAN, LIN, FlexRay, and MOST are used as in-vehicle communication technologies[4]. Although Ethernet is a relatively new to the automotive domain, it offers several desirable properties such as high bandwidth, interoperability, robustness, low cost and seamless integration with the TCP/IP stack[4]. Because it is a peer-to-peer network with full duplex communication, each ECU can communicate with one another at 100 Mb/s bandwidth. However, in order to meet the delay requirements, complementary technologies such as AVB for in-vehicle communication are required.

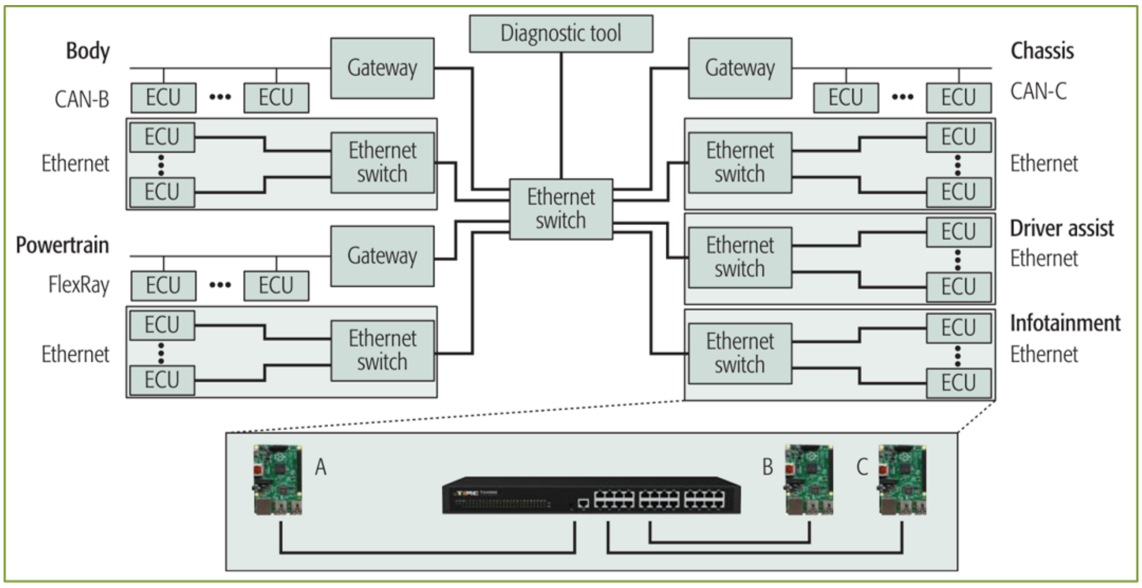


Figure 2.1: Ethernet as a backbone for in-vehicle communication[4]

Figure 2.1 is an example of an in-vehicle network with Ethernet as a backbone. Low-speed networks such as CAN, LIN, and Flexray are connected to Ethernet via a switch using gateways[4]. This facilitates the establishment of a link between ECUs that are integrated with Ethernet as the native protocol and have higher bandwidth requirements for their applications, allowing message sharing across domains. With the introduction of IoT in the automotive industry, a switched Ethernet network serves as the foundation technology for implementing V2X communication.

2.4 Service Oriented Architecture

Automotive Ethernet has resulted in a paradigm shift in the development of automotive systems. Because scalability is one of the primary advantages of using Automotive Ethernet, newer protocols and technologies that provide smooth, flexible, and scalable software solutions are required. SOA is one of the tried-and-true web services technologies that can be applied in the automotive context to support the growing complexity of automotive software. Given the resource limitations of ECUs, the SOA protocols designed for high level machines and servers cannot be directly used for the automotive software. In order to bridge the gap and reuse the concepts of the existing SOA model, new protocols are needed to be developed. AUTOSAR provides a standardized specifications for a protocols such as SOME/IP and DDS which works by incorporating the concepts of the SOA model. The details of these technologies are discussed in the following section.

2.5 Middleware in the automotive domain

The increasing complexity in the automotive applications means there is a need for standardized middleware that can provide common services and common interfaces to the application software components[8]. Middleware is a software layer that connects and manages application components running on distributed hosts[7]. In practice, a middleware is composed of a collection of existing communication protocols and carmarker-specific layers[8]. In the context of communication control, the middleware's role is to provide a layer of abstraction between the application software and the network.[11]. AUTOSAR has standardized specifications for middleware such as SOME/IP and DDS[8]. Apart from this several other middleware based on CAN protocol such as eSOC[8] are available for use in the automotive context.

2.5.1 SOME/IP

"Scalable service Oriented MiddlewarE over IP" abbreviated SOME/IP represents a middleware that was created for automotive use cases [5]. The compatibility with AUTOSAR was a necessity regarding SOME/IP at least on wire-format level [5]. SOME/IP communication is an exchange of messages between different devices like ECUs over IP [5]. The SOME/IP protocol aims to define a uniform middleware for IP-based communication within vehicles. Figure 2.2 represents the organization of the SOME/IP middleware in the ISO/OSI model.

The protocol builds on top of an existing TCP/UDP stack, adding another level of abstraction for application communication by enabling locality transparency. This property denotes the fact that an application has no knowledge of which network node provides the desired function or information. If the desired function or information is available on the same ECU, a local connection is established between

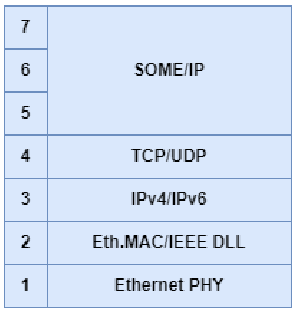


Figure 2.2: Simplified ISO/OSI model of Automotive Ethernet stack for communication control

the software components [11]. When the information, on the other hand, is on another node on the same network, the middleware performs the necessary network communication and provides the data to the application.

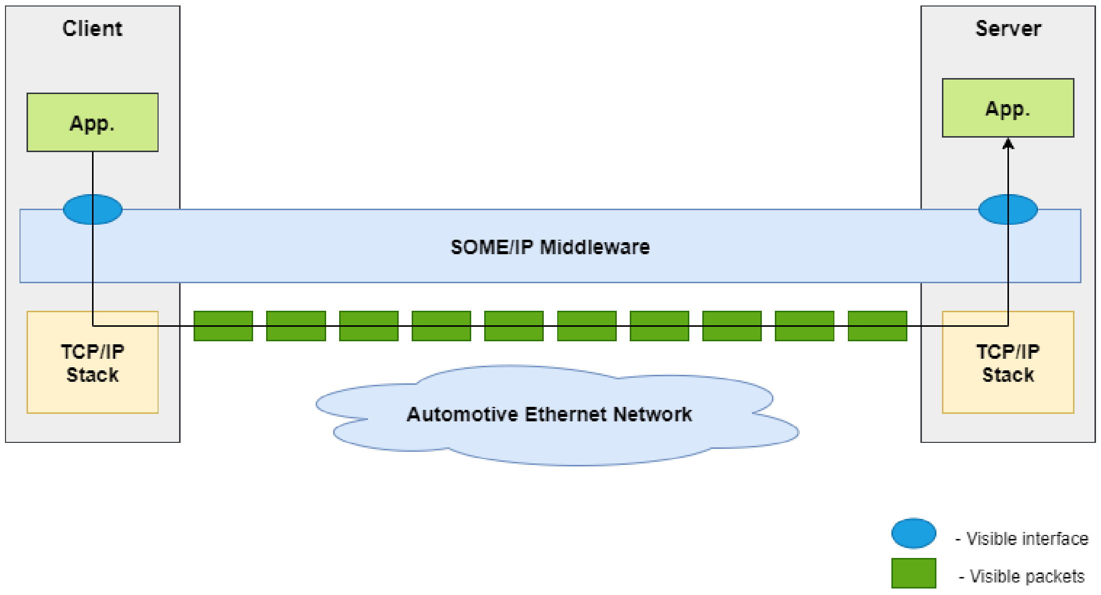


Figure 2.3: SOA representation with SOME/IP middleware

Figure 2.3 depicts a high-level overview of the SOME/IP transformer. It is based on the client-server mechanism used in service-oriented architecture. This enables a variety of methods for sending and requesting data between the client and server via RPCs and Publish-Subscribe models. The following section describes the specifics of these communication methods.

SOME/IP Message

The SOME/IP header format is depicted in Figure 2.4. The header is a 16-byte field. The message identifier is contained in the first four bytes. The message's length is represented by the next four bytes. In the third four bytes, the request identifier is placed. The SOME/IP protocol version is stored in the 12th byte. The service interface major version is stored in the 13th byte. The message type is stored in the 14th byte and indicates whether the message is a notification, a request, a response, or a request message with no return. The message error codes are stored in the header's 15th byte. The content of a serialized method, event, or field is contained in the SOME/IP payload.

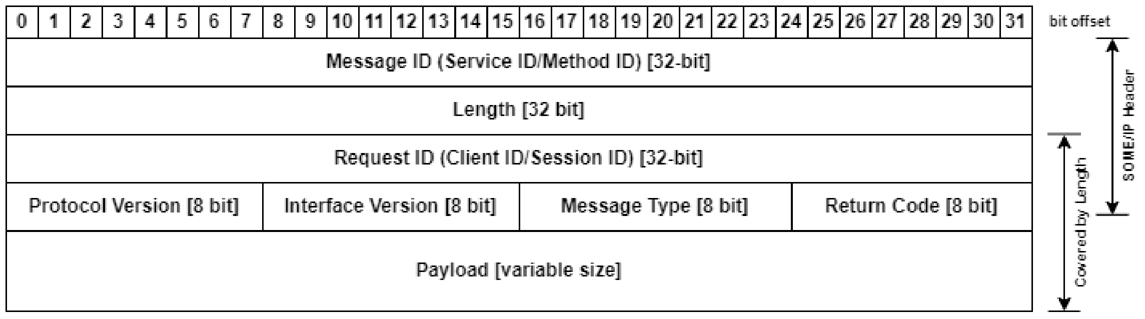
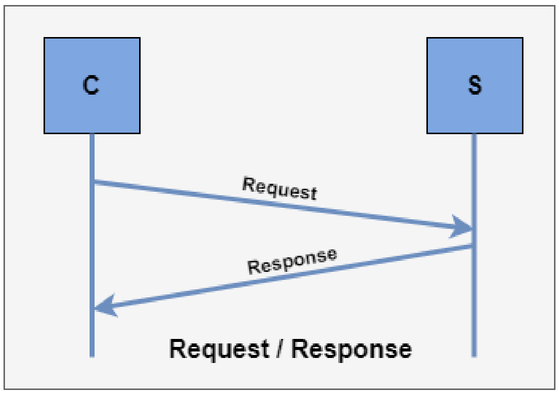
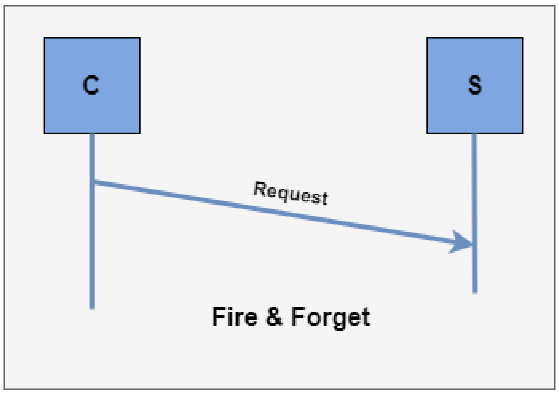


Figure 2.4: SOME/IP header format[9]

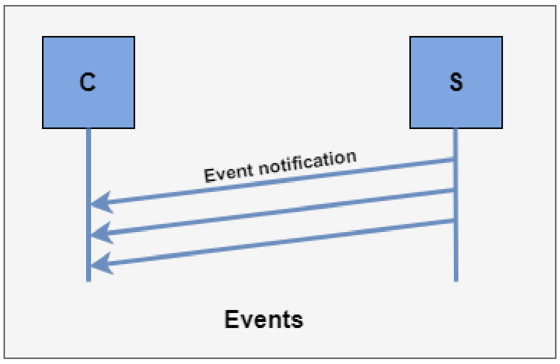
Communication methods



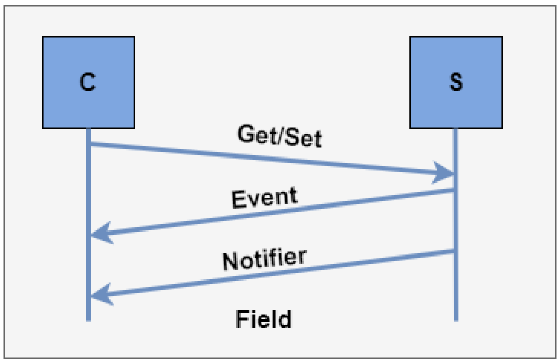
(a) “Request-Response”’-RPC: method call with response message



(b) “Fire & Forget”’- RPC: Method call without a response



(c) “Publish-Subscribe - Events”’: Event notifications



(d) “Fields”’: Set or read out the data fields of another service.

Figure 2.5: SOME/IP communication types between clients and servers

3 Implementation

To demonstrate the use of the SOME/IP technology, several devices (target controllers) are connected within a network and communication is established between them using Ethernet. In this chapter, the requirements to visualize the technology are explained and also the procedure to setup the demonstrator is discussed in detail.

3.1 Concept

Figure 3.1 depicts the physically interconnected hardware in the hardware setup. The prototype is made up of a computer running a virtual machine[18], a Raspberry Pi 3b+ [19], and an Odroid XU4[20]. A routing device connects the devices to the same network via ethernet cables. Each of the target hardware have different target architecture. The overall objective is to showcase the technology by running it simultaneously on several architectures. These devices represent the target ECUs within a vehicle that are responsible for performing specific functions based on information exchanged with SOME/IP as the underlying technology.

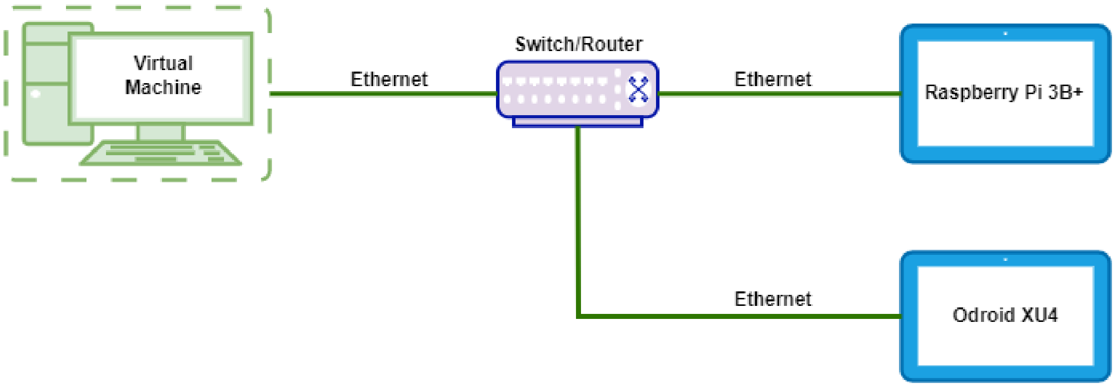


Figure 3.1: Visual representation of the hardware setup

In order to implement the applications to visualise the usage of technologies several open source stacks such as scapy-someip [13], the GENIVI vsomeip stack [12], Rust based SOME/IP implementation [14] were investigated. COVESA's (formerly known as GENIVI's) vsomeip stack appeared to be the most appropriate of the currently available implementations for this activity as it is based on POSIX and uses C++ programming language for the implementation. With the trend toward using Adaptive AUTOSAR[15] in application software development, it provides as a major motivation to realize the concepts in a POSIX-based environment.

Figure 3.2 shows the software stack and the endpoints in each of the hardware setup. A virtual machine is setup in the windows PC to emulate a Linux based environment. The devices Raspberry Pi 3b+ and Odroid XU4 are also setup with a Linux operating system. Further, these devices are installed with certain software and libraries that are required to setup the environment to demonstrate the technology. The GENIVI vsomeip stack is used to build the target applications on each of the devices. There can be multiple SOME/IP applications running on these devices. However, only one routing manager is allowed per device. The role of the routing manager is to help in routing the incoming and outgoing messages to the appropriate destinations. With access to the TCP/UDP endpoints, the devices can communicate with each other via Ethernet as shown in the figure 3.2.

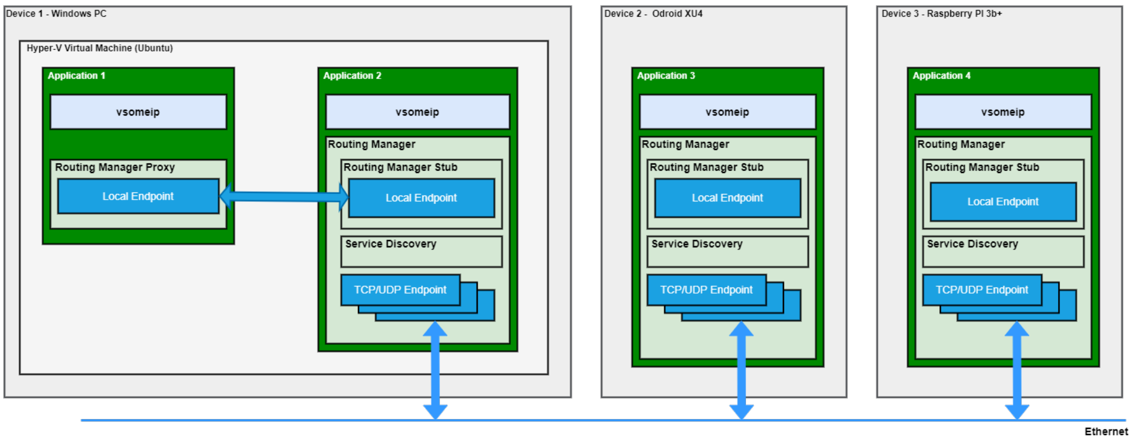


Figure 3.2: SOME/IP concept

3.2 Target Hardware

3.2.1 Virtual Machine

The Hyper-V virtualization platform from Microsoft is used to host POSIX-based environments on a Windows PC. Alternatively, other virtualization platforms can also be used for the same purpose. Based on the license constraints and availability, Hyper-V has been chosen for this project. An i386 (x86) architecture compatible OS is required to be hosted based on the concept design. For this activity, the open source Linux OS Ubuntu 20.04 LTS has been chosen for hosting on this platform. In this virtual environment, the necessary software for demonstrating the use of the technology is then installed. The physical Ethernet port on the PC is mapped to the virtual machine in order for it to communicate with the other target hardware.

3.2.2 Raspiberry Pi 3b+

The Raspberry Pi 3b+ is a single-board computer (SBC) with an ARM cortexA53 processor that is commonly used for home automation projects and prototyping. It has 1GB LPDDR2 SDRAM as well as built-in Gigabit Ethernet support. It also supports the installation of a POSIX-based environment on the board, making it appropriate for this project. For demonstration purposes, a lightweight 64-bit DietPi OS along with relevant library packages are setup on this device.

3.2.3 Odroid XU4

Odroid XU4 is an energy-efficient powerful ARM-based computing hardware that can run a variety of Linux operating systems, including Ubuntu 18.04 and Android 7.1 Nougat. It has 2GB of LPDDR3 RAM as well as Gigabit Ethernet interface support, which is required to implement SOME/IP communication concepts. As a result, this makes for good prototyping hardware because the target library packages can be configured to run the target applications on this device.

3.3 Software

3.3.1 Target Libraries

SOME/IP application development necessitates the use of several prerequisite stacks and libraries. This section contains a list of the required libraries as well as a brief overview of the libraries. All of the libraries used in this project are open source and can be used commercially. These libraries must be built for each target hardware in order for the applications to run on the devices. In this project, the target libraries are cross-compiled on the virtual machine, and the binaries are then copied to each of the devices. The following section goes over the specifics of cross-compilation.

GENIVI vsomeip stack

CommonAPI

CommonAPI C++ is a standardized C++ API specification for the development of distributed applications which communicate via a middleware for interprocess communication[16]. The main intention is to make the C++ interface for applications independent from the underlying IPC stack[16]. This library is a prerequisite in order to install vsomeip libraries. The installation process is simple and requires CMake build system for compilation.

Boost

Boost is an open source portable C++ source libraries intended to be widely useful and usable across spectrum of applications[17]. Several library functions within vsomeip stack are built based on the boost libraries and are a prerequisite for the smooth running of vsomeip stack based applications.

3.3.2 Tools

Qt Creator

Qt Creator is a cross-platform integrated development environment (IDE) built for the maximum developer experience[21]. Qt Creator runs on Windows, Linux, and macOS desktop operating systems, and allows developers to create applications across desktop, mobile, and embedded platforms[21]. The GUI for the SOME/IP-based applications is built for demonstration purposes using the open source version v5.0 of the Qt creator.

CMake

CMake is a tool to manage building of source code[22]. CMake is widely used for the C and C++ programming languages, but it can also be used to generate source code for other languages[22]. Originally intended as a generator for various dialects of Makefile, CMake now generates project files for modern buildsystems such as Ninja as well as project files for IDEs such as Visual Studio and Xcode[22]. CMake is used as the build system in this project for compilation as well as cross-compilation of library packages and SOME/IP applications for different target hardware.

PuTTY

PuTTY is an SSH and telnet client, developed originally by Simon Tatham for the Windows platform[23]. PuTTY is open source software that is available with source code and is developed and supported by a group of volunteers[23]. This tool is required to run the terminals for Raspberry Pi3b+ and Odroid XU4 remotely on the host machine. This enables to smoothly switch between the target devices when running the applications on the target hardware respectively.

3.4 Cross-compilation

Every development board is embedded with a specific amount of RAM, storage capacity, input and output peripherals and other hardware components. Hosting the target environment on multiple boards can be complicated and time consuming. Furthermore, building target libraries on these boards can take a long time and may fail in some cases. To address these issues, it is worthwhile to setup a generic build environment on a single platform and build the projects for different targets accordingly. This process is called as cross-compilation. In this section, the process to setup a cross-compilation environment on the Linux platform is illustrated and along with it, the procedure to cross-compile boost libraries for Raspberry Pi and Odroid XU4 target platforms is demonstrated respectively.

3.4.1 Installing cross compilers on the host machine

In this section, the basic requirements to setup a cross-compilation tool-chain is described. Also, based on the requirements for the demonstration of the SOME/IP technology, build process for libraries such as Boost, CommonAPI, vsomeip and other relevant libraries are described in detail.

In order to cross-compile, appropriate tool-chain packages has to be first setup in the host environment. The commands from the following listings are required to be run in a terminal window in the Linux machine. Please note that an active internet connection is required to download the packages from the server.

```
user@machine:~$ sudo apt-get install gcc-arm-linux-gnueabihf
g++-arm-linux-gnueabihf
```

Listing 3.1: Command to install packages for ARM 32-bit (armv7) tool-chain

```
user@machine:~$ sudo apt-get install gcc-aarch64-linux-gnu
g++-aarch64-linux-gnu
```

Listing 3.2: Command to install packages for ARM 64-bit(armv8) tool-chain

```
user@machine:~$ sudo apt-get install build-essential
manpages-dev openjdk-8-jdk libssl-dev wireshark
g++-aarch64-linux-gnu
```

Listing 3.3: Command to install other required packages

3.5 Demo Application

In this section, a demo application built with Qt Creator is shown to demonstrate the technology. Two separate applications are created, one for the server ECU and one for the client ECU. These user interface-based applications are used to demonstrate intra-ECU and inter-ECU communication. However, when inter-ECU communication is required, the UI-based application is only run on the virtual machine. The SOME/IP applications are run without the UI on the other target hardware. In the case of inter-ECU communication, for example, when the server application is run on the virtual machine, it is represented by a UI-based application, whereas the client application is run on one of the target hardware without a graphical interface. Each SOME/IP-based communication method is represented on both the client and server applications, with detailed information displayed in a terminal window. The responses received by the client are displayed on the terminal window, which also contains detailed information about the messages sent and received. The specifics of the implementations are covered in the following sections.

3.5.1 vsomeip application

To create a vsomeip application, the libraries described in the preceding sections must be available. To realize the SOME/IP communication types, an application must be written with information containing details such as the name of the application, the services offered, and the services to be requested. A registry handler function must be used to register the services and service instances in a registry. In addition, the application's requested services must be stored in the availability handler registry using the availability handler functions. The vsomeip stack provides APIs for creating a vsomeip application and registering services in the appropriate registry. The code example in the listing 3.4 shows a snippet of code from the server application related to the creation of an application and storing information about offered and requested services in the registry handlers. The RPM service denotes the method to create a service that offers request-response type of service. To offer services which supports event notifications, the events have to be inserted in an event group. This is demonstrated using the indicator service. To request a service offered by other vsomeip applications, a service request has to be invoked that includes the service information details.

```
#include <vsomeip/vsomeip.hpp>

const char ApplicationName[] = "vsomeip_server";

int main()
{
    //Create a vsomeip application
    app = vsomeip::runtime::get()->create_application(
       ApplicationName);

    bool Is_init_Successful = app->init();

    if(Is_init_Successful)
    {
        /*  RPM Service registration
        Macros : RPM_SERVICE_ID 0x2000, RPM_INSTANCE_ID 0x2100,
           RPM_METHOD_ID 0x2200  */
        //message handler registration. Callback function name:
           on_message
        app->register_message_handler(RPM_SERVICE_ID,
           RPM_INSTANCE_ID, RPM_METHOD_ID, on_message);
        //Offer the service to the clients
        app->offer_service(RPM_SERVICE_ID, RPM_INSTANCE_ID);

        /*Indicator service registration
        Macros : INDICATOR_SERVICE_ID 0x2500, INDICATOR_INSTANCE_ID
            0x2510, INDICATOR_METHOD_ID 0x2520,
        INDICATOR_EVENTGROUP_ID 0x4400, INDICATOR_EVENT_ID 0x4300
           */
        app->register_message_handler(INDICATOR_SERVICE_ID,
           INDICATOR_INSTANCE_ID, INDICATOR_METHOD_ID, on_message);
        app->offer_service(INDICATOR_SERVICE_ID,
           INDICATOR_INSTANCE_ID);
        its_groups.insert(INDICATOR_EVENTGROUP_ID);
        app->offer_event(INDICATOR_SERVICE_ID,
           INDICATOR_INSTANCE_ID, INDICATOR_EVENT_ID, its_groups,
           true);

        /*Temperature service registration
        Macros : TEMP_SERVICE_ID 0x2500; TEMP_INSTANCE_ID 0x2510;
           TEMP_METHOD_ID 0x2520 */
        app->register_message_handler(TEMP_SERVICE_ID,
           TEMP_INSTANCE_ID, TEMP_METHOD_ID, on_message);
        //Availability handler registration. Callback function name
           : on_availability
        app->register_availability_handler(TEMP_SERVICE_ID,
           TEMP_INSTANCE_ID, on_availability);
        //Request temperature service from other server/client
        app->request_service(TEMP_SERVICE_ID, TEMP_INSTANCE_ID);

        //Start the application
        app->start();
    }
    else
    {
        //Error message output if application is unable to start
        std::cerr << "Error Code: VS001: Failed to start
           application: " <<  ApplicationName  << " due to
           unsuccessful initialization " << std::endl;
    }
}
```

Listing 3.4: vsomeip application initial configurations

The callback function on_message is demonstrated in listing 3.5. A generic callback function can be used to implement all of the services that are offered and requested, or multiple callback functions can be implemented based on the intended functionality upon the reception of a message from a specific service. In this example listing, a generic callback function is implemented, and the corresponding function call is invoked to provide the intended functionality based on the service ID extracted from the response message header.

```
void on_message(const std::shared_ptr<vsomeip::message> &_RespMsg)
{
  //Get the service ID of the response message
  int ResponseServiceID = (int)_RespMsg->get_service();

  switch(ResponseServiceID)
  {
    //Invoke function related to rpm service
    case RPM_SERVICE_ID: on_rpm_service_Msg(_RespMsg); break;
    //Invoke function related to speed service
    case SPEED_SERVICE_ID: on_speed_service_Msg(_RespMsg); break;
    //Invoke function related to temperature service ID
    case TEMP_SERVICE_ID: on_temp_service_Msg(_RespMsg); break;
    //Invoke function related to fuel service ID
    case FUEL_SERVICE_ID: on_temp_service_Msg(_RespMsg); break;
    //Invoke function related to fuel service ID
    case INDICATOR_SERVICE_ID: on_indicator_service_Msg(_RespMsg);
       break;
    case default: break;
  }
}
```

Listing 3.5: vsomeip message handler callback

The listing 3.6 shows how to process a received message for RPM service. Once the message has been received, the payload length, sender, message type, and other information can be extracted from the message header. Furthermore, the received data can be processed in order to compute and retrieve the RPM data that is available on the server side. The header information for a response message can be created and sent back to the client based on the type of message received. Data shall not be sent back to the client in the event of a fire and forget communication type. Similarly, messages for other services are processed, and responses are returned as appropriate.

```
on_rpm_service_Msg(const std::shared_ptr<vsomeip::message> &
    _RespMsg)
{
  //extract the payload from the received response
  std::shared_ptr<vsomeip::payload> its_payload = _RespMsg->
      get_payload();
  //Get the length of the payload
  vsomeip::length_t payload_len = its_payload->get_length();

  int ResponseMessageType =(int)_response->get_message_type();

  int ResponseServiceID = (int)_response->get_service();

  std::stringstream RecievedResponse;

  //Check if recieve response is related to RPM service
  if((RPM_SERVICE_ID == ResponseServiceID) && (0x00 ==
      ResponseMessageType))
  {
      for (vsomeip::length_t i=0; i < payload_len; i++)
      {
          RecievedResponse << std::setw(2) << std::setfill('0') <<
              std::hex
          << (int)*(its_payload->get_data()+i) << " ";
      }
      //Send the current RPM data from the server
      (void)SendRpmData(_response);
  }
  else
  {
    //Place holder for other response message types
  }
}
```

Listing 3.6: Example of received message processing

The availability handler callback function is shown in listing 3.7. This function is called during the application's initialization phase and remains active throughout. Once the service becomes unavailable, the availability stated also is inverted accordingly. Based on the information stored in the service registry, the vsomeip stack determines whether a service and its instances are available. The end user can be notified if a specific service is available for consumption using the information. In addition, depending on the availability, internal back-end computations related to the service can be started in order to generate the necessary data. Several utility functions are implemented in this demonstration to invoke and compute the required information once a service and its instance are available. Listing 3.8 is a snippet of an RPM service implementation. When the RPM service becomes available, a thread associated with the RPM service is notified in order to compute the necessary data.

```
void on_availability(vsomeip::service_t _service, vsomeip::
    instance_t _instance, bool _is_available)
{
    bool IsServiceAvailable_b = util_IsServiceAvailable(_service,
        _is_available);

    std::cout << "CLIENT-> ["<< ApplicationName << "]" <<": Service
        ["
            << std::setw(4) << std::setfill('0') << std::hex <<
                _service << "." << _instance
            << "] is "
            << (IsServiceAvailable_b ? "available." : "unavailable.
                [Error Code: VS003]")
            << std::endl;
}
```

Listing 3.7: vsomeip availability handler callback function

```
bool util_IsServiceAvailable(const vsomeip::service_t _service,
    const bool _is_available)
{
    bool IsServiceAvailable_b = false;
    switch(_service)
    {
      case RPM_SERVICE_ID :
        if(_is_available)
        {
            IsServiceAvailable_b = true;
            //Notify the thread related to RPM service once the
                service ID is available
            (void)RPMRequestThreadNotify();
        }
```

```
            break;
        }
        return IsServiceAvailable_b;
}
```

Listing 3.8: availability handler utility function

3.5.2 Server Application

The server user interface is depicted in Figure 3.3. Several different types of information are sent and received between the server and the client for demonstration purposes. The server sends the RPM, Speed, Fuel, and Indicator information to the client. The data for RPM, Speed, and Fuel can all be changed by using the vertical scroll bars. The progress bars represent the information that is dynamically updated. The Request-Response communication type is represented by the RPM information. It means that whenever the client makes a request, the server sends the most up-to-date RPM information to the client. Similarly, the Speed and Fuel information represent field-based event notifications from the server to the client. The information for the left and right indicators can be changed by using the respective buttons on the user interface. Whenever the value is updated, the data is sent out immediately with message type information set to “MT_NOTIFICATION”. The information is sent as event notifications from the server sent to the client. A notification message with service identifier, instance identifier, the client identifier, message type, payload and other details is packed and triggered from the server. The TEMP element indicates the ambient temperature. The data is sent as a request message from the client to the server, with no expectation of a response message from the server (fire & forget communication type). The temperature data is stored in the server application and can be used for further processing.

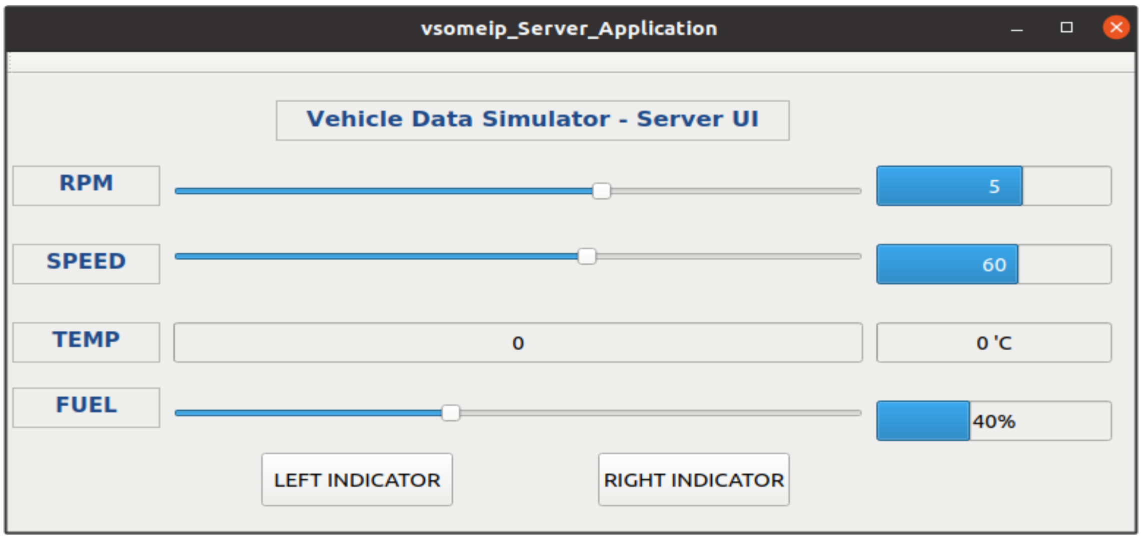

vsomeip_Server_Application
Vehicle Data Simulator - Server UI
RPM
5
SPEED
60
TEMP
0
0 'C
FUEL
40%
LEFT INDICATOR
RIGHT INDICATOR

Figure 3.3: Qt based Server application GUI

3.5.3 Client Application

The client user interface is depicted in Figure 3.4. Similar to the server user interface, the client user interface consists of graphical elements that represent the information related to RPM, speed, temperature and fuel data. The information on the user interface is dynamically updated as and when the data is received from other vsomeip applications. However, the data for RPM is received only when the "Request RPM" button is pressed indicating a request for information being triggered to the server. The message type information is set as "MT_REQUEST" indicating a request is placed to the server application. The server application then provides a response with the response message type set to "MT_RESPONSE". In addition, the client sets the ambient temperature, and the data is sent to the server whenever the "Send Current Temp" button on the user interface is pressed. This data represents the fire & forget communication type, in which the client pushes data to the server without requesting acknowledgement from the server. This message is received by the receiver application with the type information "MT REQUEST NO RETURN", indicating that no further response to the sender is required.

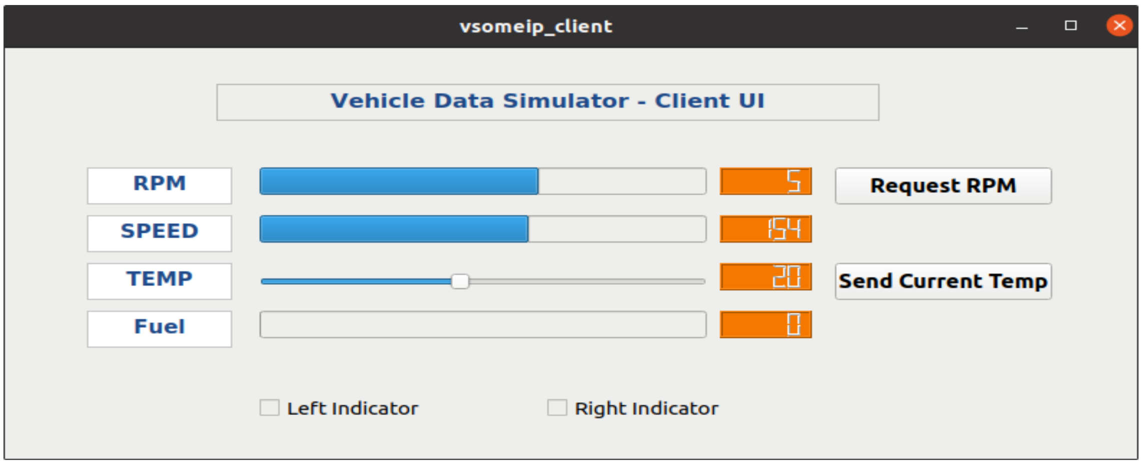


Figure 3.4: Qt based Client application GUI

3.5.4 Comunication establishment between devices

The vsomeip stack uses the json format to represent the configuration parameters that are passed to the applications during startup. This configuration file is required when communication between applications running on multiple target hardware is required. This file contains the MAC address, the addressing type, i.e. whether it is a unicast or a multicast addressing scheme, the service IDs, instance IDs, event and event group IDs. It also includes the details of the service discovery, as well as the transport layer (TCP or UDP) and the application name used for the applications. In addition, within the configuration file, the configuration for logging trace data can be enabled or disabled. Before beginning, the proper environment must be created. To successfully establish communication, the proper environment must be set before running the application, and the path to the configuration file must be specified. Listing 3.9 shows a small portion of the configuration file used for this project. The application ID can be assigned as per the user choice and has to be unique for each of the applications. The configuration file can be further scaled with additional parameters that are listed in the vsomeip documentation[24].

```
{
   "unicast" : "192.168.0.103",
   "netmask" :"255.255.255.0",
      "logging" :
      {
         "level" : "debug",
         "console" : "true",
         "file" : { "enable" : "true", "path" : "/tmp/vsomeip.log"
            ↪ },
         "dlt" : "false"
      },
      "applications" :
         [
         {
            "name" : "Demo_Application",
            "id" : "0x1212"
         }
   ],
      "services" :
         [
         {
            "service" : "0x2000",
            "instance" : "0x2100",
            "unreliable" : "30509"
         }
   ],
      "routing" : "vsomeip_server",
      "service-discovery" :
      {
         "enable" : "true",
         "multicast" : "224.224.224.245",
         "port" : "30490",
         "protocol" : "udp",
         "initial_delay_min" : "10",
         "initial_delay_max" : "100",
         "repetitions_base_delay" : "200",
         "repetitions_max" : "5",
         "ttl" : "3",
         "cyclic_offer_delay" : "2000",
         "request_response_delay" : "1500"
      }
}
```

Listing 3.9: example of vsomeip configuration file - vsomeip.json

4 Results

Enter text here

4.0.1 Server Application Output

4.0.2 Client Application Output

4.0.3 Troubleshooting guide

5 Conclusion

Conclusion

Bibliography

[1] Kozierok, C. M., Correa, C., Boatright, R. B., & Quesnelle, J. (2014). Automotive Ethernet: The Definitive Guide. Intrepid Control Systems.

[2] Zhang, H., Pan, Y., Lu, Z., Wang, J., & Liu, Z. (2021). A Cyber Security Evaluation Framework for In-Vehicle Electrical Control Units. IEEE Access, 9, 149690-149706.

[3] Hank, P., Müller, S., Vermesan, O., & Van Den Keybus, J. (2013, March). Automotive ethernet: in-vehicle networking and smart mobility. In 2013 Design, Automation & Test in Europe Conference & Exhibition (DATE) (pp. 1735-1739). IEEE.

[4] Han, S., & Kim, H. (2016). On AUTOSAR TCP/IP performance in in-vehicle network environments. IEEE Communications Magazine, 54(12), 168-173.

[5] A. Ioana and A. Korodi, ”VSOMEIP - OPC UA Gateway Solution for the Automotive Industry,” 2019 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC), 2019, pp. 1-6, doi: 10.1109/ICE.2019.8792619.

[6] G. L. Gopu, K. V. Kavitha and J. Joy, ”Service Oriented Architecture based connectivity of automotive ECUs,” 2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT), 2016, pp. 1-4, doi: 10.1109/ICCPCT.2016.7530358.

[7] Park, J., Kim, S., Yoo, W., & Hong, S. (2006, October). Designing real-time and fault-tolerant middleware for automotive software. In 2006 SICE-ICASE International Joint Conference (pp. 4409-4413). IEEE.

[8] Navet, N., Song, Y., Simonot-Lion, F., & Wilwert, C. (2005). Trends in automotive communication systems. Proceedings of the IEEE, 93(6), 1204-1223.

[9] “Some/IP protocol specification - AUTOSAR.” [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/foundation/1-0/AUTOSAR_PRS_SOMEIPProtocol.pdf. [Accessed: 11-Oct-2021].

[10] “Example for a Serialization Protocol (SOME/IP),” AUTOSAR Release 4.3.1, 31-Mar-2014. [Online]. Available: https://www.autosar.org/fileadmin/

user_upload/standards/classic/4-1/AUTOSAR_TR_SomeIpExample.pdf. [Accessed: 06-Nov-2022].

[11] Rumez, M., Grimm, D., Kriesten, R., & Sax, E. (2020). An overview of automotive service-oriented architectures and implications for security countermeasures. IEEE access, 8, 221852-221870.

[12] COVESA, "Covesa/vsomeip at 2.14.16," GitHub. [Online]. Available: https://github.com/COVESA/vsomeip/tree/2.14.16. [Accessed: 14-Mar-2022].

[13] Jamores, "Jamores/ETH-Scapy-someip: Automotive ethernet some/IP-SD scapy protocol," GitHub. [Online]. Available: https://github.com/jamores/eth-scapy-someip. [Accessed: 12-Nov-2021].

[14] de Almeida, J. F. B. (2020). Rust-based SOME/IP implementation for robust automotive software.

[15] AUTOSAR development cooperation, "Adaptive platform," AUTOSAR, 06-Dec-2021. [Online]. Available: https://www.autosar.org/standards/adaptive-platform/. [Accessed: 18-Dec-2021].

[16] Covesa, "Releases · COVESA/CAPICXX-core-tools," GitHub. [Online]. Available: https://github.com/COVESA/capicxx-core-tools/releases?after=3.1.10.1. [Accessed: 06-Jan-2021].

[17] Boost C++ libraries. [Online]. Available: https://www.boost.org/users/history/version_1_65_0.html. [Accessed: 10-Jan-2022].

[18] Scooley, "Introduction to hyper-V on Windows 10," Microsoft Docs. [Online]. Available: https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/. [Accessed: 12-Nov-2021].

[19] "Raspberry Pi 3 Model B+.," [Online]. Available: https://datasheets.raspberrypi.com/rpi3/raspberry-pi-3-b-plus-product-brief.pdf. [Accessed: 04-Nov-2021].

[20] "Odroid XU4 user Manual - Odroid Magazine." [Online]. Available: https://magazine.odroid.com/wp-content/uploads/odroid-xu4-user-manual.pdf. [Accessed: 08-Nov-2021].

[21] "Embedded Software Development Tools: Cross platform IDE: Qt creator," Embedded Software Development Tools — Cross Platform IDE — Qt Creator. [Online]. Available: https://www.qt.io/product/development-tools. [Accessed: 06-Dec-2021].

[22] "CMake Reference Documentation," CMake Reference Documentation - CMake 3.23.0 Documentation. [Online]. Available: https://cmake.org/cmake/help/latest/. [Accessed: 12-Dec-2021].

[23] "Download Putty - a free SSH and telnet client for windows," Download PuTTY - a free SSH and telnet client for Windows. [Online]. Available: https://www.putty.org/. [Accessed: 17-Mar-2022].

[24] Covesa. (2020, February 12). Vsomeip/vsomeipuserguide at master · COVESA/vsomeip. GitHub. Retrieved February 2, 2022, from https://github.com/COVESA/vsomeip/blob/master/documentation/vsomeipUserGuide