Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis

Definition and Evaluation of an Onboard Vehicle API Concept

Rachit Garg

Course of Study:	M.Sc. Information Technology Specialization Embedded Systems Engineering
Examiner:	Dr. Ilche Georgievski
Supervisor:	Dipl. Infom. Jens Kübler, Vector Informatik GmbH

Commenced:	May 03, 2023
Completed:	November 03, 2023

Acknowledgements

The satisfaction and joy of learning technologies that have the potential to shape our future and possibly the current time by doing is just amazing and I am grateful to get this opportunity and want to express my gratitude to Dr. Ilche Georgievski for letting me pursue my Master Thesis at the Institute of Architecture of Application Systems, University of Stuttgart.

The work would not have been possible without the constant support and mentoring by my supervisor Dipl. Infom. Jens Kübler at Vector Informatik GmbH. His timely feedback and thought process helped me not only learn new technologies, but also push myself to the limits to become a better Software Engineer and transition into the professional world.

Finally I would like to thank the whole staff of INFOTECH for an awesome study program at the University of Stuttgart, my fellow workmates at Vector Informatik GmbH and of course my friends and family, who always believed in me.

Abstract

As the world witnesses an era of Software Defined Vehicle (SDV) and Internet of Things (IoT), the vehicles start becoming more intelligent, bringing in more software complexity than ever. This includes the Electrical/Electric (E/E) systems embedded inside today's SDVs. The Electronic Control Unit (ECU) is now capable of running software which can not only communicate with other ECUs within the vehicle, but also to the outside world using protocols which are already in use within the Information Technology (IT) world for a long time. This lead to the emergence of an off-board vehicle Application Programming Interface (API) concept which allows external devices like smartphones or servers to access vehicle information such as those of sensors and other peripherals inside the vehicle. The Connected Vehicle Systems Alliance (COVESA) came up with an approach of defining a catalog for vehicle signals using the Vehicle Signal Specification (VSS) initiative. The World Wide Web Consortium (W3C) then brings this into use by defining Vehicle Information Service Specification (VISS). But the current VISS standard, namely version 2, defines transport bindings for WebSocket, Hypertext Transfer Protocol (HTTP) and Message Queuing Telemetry Transport (MQTT). These are well established protocols in the IT world and also suitable for off-board use cases. But if we want to use such an API concept inside the vehicle, so that the internal applications can also take advantage of a standardized catalog of signals and be reused between different vehicles, we need to consider protocols already in use inside vehicles as well. Scalable service-Oriented MiddlewarE over IP (SOME/IP) is one such example. Therefore, in this thesis, we elaborate the concept of a vehicle API, look at related technologies in more detail. Later, we take a look at one of the suitable candidates for an in-vehicle protocol, namely SOME/IP and see it's respective trade-offs. Finally, we conceptualize and realize a Vehicle API concept for onboard usage.

Contents

List of Figures

List of Tables

Listings

Acronyms

ADAXO Automotive Data Access - Extended and Open. 40

API Application Programming Interface. 4

ASCII American Standard Code for Information Interchange. 32

AUTOSAR AUTomotive Open System ARchitecture. 41

BOM Byte Order Mark. 46

CAN Controller Area Network. 41

COVESA The Connected Vehicle Systems Alliance. 4

DDS Data Distribution Service. 17

DoIP Diagnostics over Internet Protocol. 13

dp data point. 51

E/E Electrical/Electric. 4

ECU Electronic Control Unit. 4

ExVe Extended Vehicle. 40

HTTP Hypertext Transfer Protocol. 4

HTTPS Hypertext Transfer Protocol Secure. 19

IAM Identity Access Management. 111

IDL Interface Definition Language. 19

IoT Internet of Things. 4

IP Internet Protocol. 13

IPC Inter Process Communication. 41

IT Information Technology. 4

IVI In-Vehicle Infotainment. 40

JSON JavaScript Object Notation. 7

JWT JSON Web Token. 20

LIN Local Interconnect Network. 41

LSB Least Significant Byte. 27

MOST Media Oriented System Transport. 41

MQTT Message Queuing Telemetry Transport. 4

MSB Most Significant Byte. 27

NEVADA Neutral Extended Vehicle for Advanced Data Access. 40

OEM Original Equipment Manufacturer. 40

OSI Open Systems Interconnection. 26

OTA Over-The-Air. 13

SD Service Discovery. 33

SDV Software Defined Vehicle. 4

SOA Service Oriented Architecture. 26

SOME/IP Scalable service-Oriented MiddlewarE over IP. 4

SOME/IP-TP SOME/IP Transport Protocol. 71

SOME/IP-SD SOME/IP Service Discovery. 6

SOVD Service Oriented Vehicle Diagnostics. 13

TCP Transmission Control Protocol. 75

TLV Type-Length-Value. 6

TTL Time To Live. 29

UDP User Datagram Protocol. 28

UTF-8 Unicode Transformation Format-8. 46

V2X Vehicle-to-everything. 13

VDA Verband der Automobilindustrie. 40

VHAL Vehicle Hardware Abstraction Layer. 41

VIN Vehicle Identification Number. 18

VISS Vehicle Information Service Specification. 4

VM Virtual Machine. 9

VSC Vehicle Service Catalog. 16

VSS Vehicle Signal Specification. 4

W3C The World Wide Web Consortium. 4

YAML Yet Another Markup Language. 18

1 Introduction

1.1 Context

Recently, the mobility sector has seen a number of technological innovations. A vehicle which was seen just as a means of reaching from point A to B, is now focusing on how to provide more immersive experience to it's passengers. Many of these innovations are possible due to a number of factors like electrification, expansion of IoT, and advancements in vehicle ECU architectures. This has lead to software becoming one of the prominent defining features inside vehicles and hence, turning them into SDVs.

Therefore, new technologies and standards are emerging in this field. These include Over-The-Air (OTA) updates, Vehicle-to-everything (V2X) [12], Vehicle API [5], and Service Oriented Vehicle Diagnostics (SOVD) [66] to name a few. These open up opportunities for the vehicles to not only consume services, but to also provide them. This means that we need to somehow connect classical Embedded Systems concepts with the technologies found in the IT world like connected and scalable cloud systems. This is a challenging task as we need to introduce connected features, keeping in mind not only the security aspects, but also the functional safety aspects [62].

These factors also started to influence the communication inside the vehicle, Making the software within ECUs increasingly complex. Hence, we see a shift from domain to zonal architecture [31], enabled with the help of Automotive Ethernet [53], consisting of Internet Protocol (IP) based application layer protocols such as SOME/IP [9] and Diagnostics over Internet Protocol (DoIP) [46].

The focus of this master thesis is on one such technology which aims to be a part of these Software Defined Connected Vehicles, namely the Vehicle API. A Vehicle API is typically built of three components: A stable data catalog accessible to applications, an operationalization in form of methods to access the data, and a network binding enabling programmed clients to obtain the data via a defined endpoint. By this, a Vehicle API enables interoperability and portability across applications and platforms. VSS solves the issue of defining a standardized vehicle catalog, whereas VISS solves the issue of operationalization and network bindings that the vehicle API will work with. With the VISS [13] [49] an open service implementation for VSS [25], three network bindings - HTTP, MQTT and WebSocket – have been defined. The network bindings are suitable for off-board applications, but it is unclear if they fulfill onboard application requirements.

1.2 Problem Statement

This master thesis aims to evaluate the state of the art in existing Vehicle API solutions with respect to automotive requirements such as performance and usability. Based on this evaluation, a suitable concept is created, implemented and evaluated. Focal points are network binding solutions and operationalization of a Vehicle API. Hence we conceptualize, implement and evaluate a concept to map VISS around one such tried and tested automotive protocol, i.e. SOME/IP and look at it's various trade-offs. Figure 1.1 shows the problem statement in an intuitive fashion.

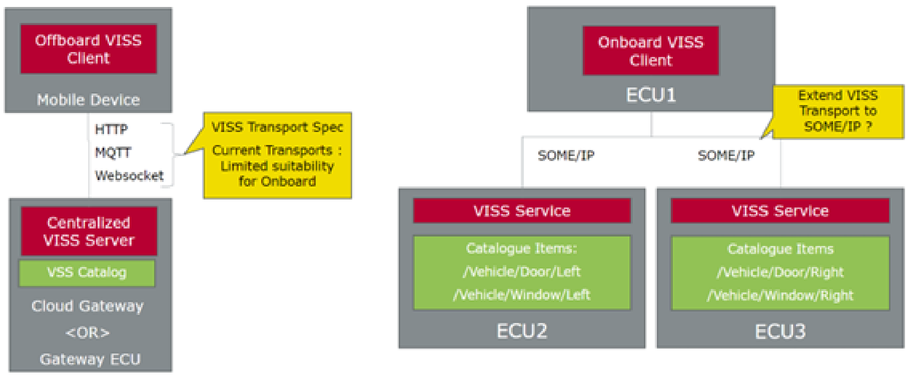


Figure 1.1: Problem Statement

1.3 Methodology

The process of the thesis was split into the following steps:

1. **Orientation:**
    - Understanding the concept of a Vehicle API and related concepts.
    - Analyzing the recent developments in the field of Vehicle API and related technologies.
    - Analyzing VSS, VISS, SOME/IP and SOME/IP-SD specifications.
2. **Conceptualization:**
    - Framing design questions.
    - Designing and comparing concepts for service discovery.
    - Designing and comparing concepts for representing methods and data to be mapped from VISS to SOME/IP.
    - Understanding whether the approaches are feasible.
3. **Realization:**
    - Choosing one or more feasible design concepts.
    - Choosing a set of libraries and frameworks to implement them.

- Choosing or developing (in case they do not exist) the classes and interfaces needed to implement VISS clients and servers.
- Integrating the chosen and developed classes and interfaces according to client and server implementation.
- Implementing a public API to access the developed client's methods.

4. **Assessment:**
   - Designing a test scenario.
   - Implementing tests to verify client's methods and Measure various parameters to evaluate the overall solution.

1.4 Document Structure

- **Chapter 2** gives a brief overview on the important technical topics mentioned throughout this report.
- **Chapter 3** highlights the recent developments and the related technologies being used and under discussion at the time of writing this report.
- **Chapter 4** discusses and gives an analysis on the design decisions and the various trade-offs that need to be considered.
- **Chapter 5** provides the necessary information on the implementation details.
- **Chapter 6** provides insight on the evaluation results including benchmarks and the testing methodology used.
- **Chapter 7** summarizes the findings, gives an outlook and concludes the thesis.

2 Background Information

This chapter describes the technologies which are relevant to understand the work done as part of this thesis. first, we describe what a Vehicle API is and it's related components which include a catalog, operationalization and a network binding. Later, we describe their concrete forms namely VSS, VISS, Automotive Ethernet along with SOME/IP.

2.1 Vehicle API

According to [5] and [22], A vehicle API provides ways to access and manipulate data inside a vehicle without having to deal with the complexity of how this data is actually generated or stored, making it easier for application programmers without the domain or vehicle model specific knowledge to leverage this data into their applications. Figure 2.1 shows the components of a Vehicle API.

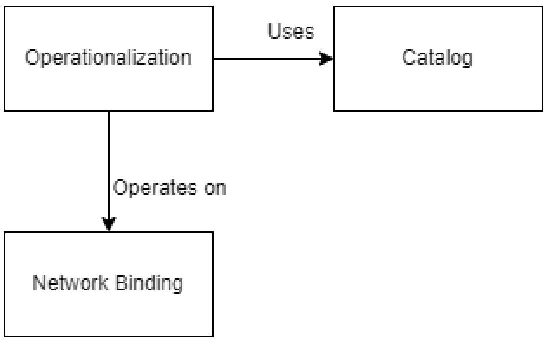


Figure 2.1: Components of a Vehicle API

The main idea is to offer a standardized catalog of vehicle signals or services, provide ways to access these via an operationalization and a network binding technology to enable the transport.

2.1.1 Catalog

A catalog can be of either services and signals. A service catalog, for example, Vehicle Service Catalog (VSC) [24] [23] aims to model complex interfaces consisting of custom data types and methods. This could include vehicle specific API definition. On the other hand, VSS focuses on defining a standard catalog for vehicle signals. Signals are data representing a specific aspect of vehicle state, for example, speed.

2.1.2 Operationalization

It describes conceptual access methods for data retrieval. It handles the (behavioral) semantics of these methods. These semantics could be for example, retrieving data (pull or push based) or handling concurrent access semantics. VISS is one such example of an operationalization for VSS.

2.1.3 Network Binding

Network Binding refers here to the application layer protocol used for the transport of an operationalization. Examples of network bindings include HTTP [58], WebSockets [55] MQTT [60], SOME/IP and Data Distribution Service (DDS) [61].

2.2 VSS

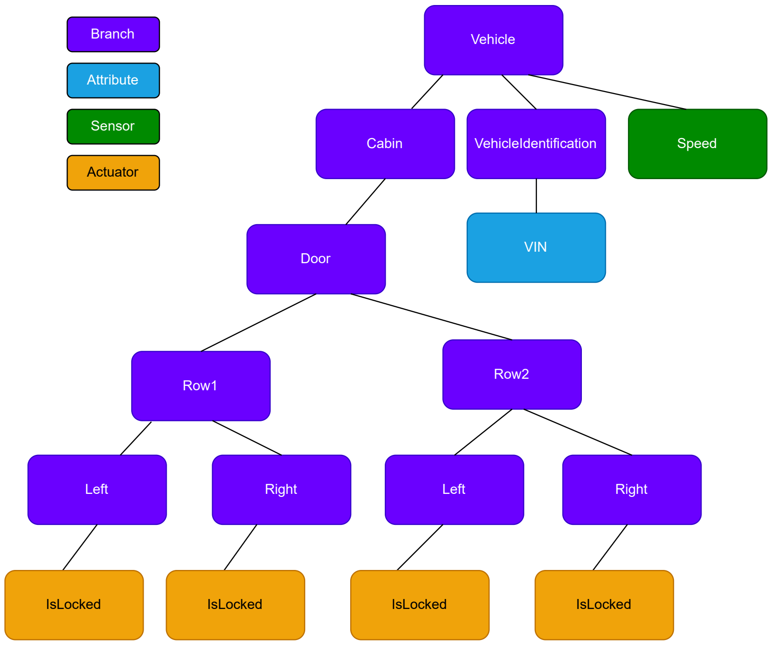


Figure 2.2: Example VSS Tree

VSS [25] by COVESA defines a syntax and a standardized catalog of vehicle signals. It defines a simple type model with a set of primitive types, arrays and in version 4 structs that one may utilize to define vehicle specific signals. Type constraints and physical units can be modeled. They may be attributed to be Sensors, Attributes and Actuators. Figure 2.2 shows an example VSS tree constructed by taking signals from the API catalog of the Mahindra model from [29].

The signals can be structured in a tree format and the nodes are divided into:

1. **Branches:** These are nodes which do not contain any value but are needed to reach the leaf nodes. By defining branches on is able to structure and instantiate the defined signals multiple times. Example: **"Vehicle.Cabin.Door.Row1.Right"** and "**Vehicle.Cabin.Door.Row2.Right"**
2. **Attributes:** These are leaf nodes containing constant values like Vehicle Identification Number (VIN). Example : **"Vehicle.VehicleIdentification.VIN"**.
3. **Sensors:** These are leaf nodes containing values which change and can be read like speed. Example **"Vehicle.Speed"**.
4. **Actuators:** These are leaf nodes containing values which can be read and written to. Example: setting the **"Vehicle.Cabin.Door.Row1.Left.IsLocked"** from "True" to "False".

For specifying various vehicle signals, vspec [18] files are used. It is an abstraction Data Definition Language written in Yet Another Markup Language (YAML) syntax. It has support for features such as includes [20] and overlays [21].

Includes allow to import signal specifications from another vspec files into the desired vspec file and also reuse same signals under different branches using prefixes. Listing 2.2 shows an example of includes where a Main.vspec file includes another vspec file named Mirrors.vspec. Let us assume this file contains a boolean signal named isHeatingOn (as shown in Listing 2.1) which indicates whether heating is on (true) or off (false) for a particular rear view mirror. Now in the Main.vspec isHeatingOn is prefixed with both the prefixes (Vehicle.Body.Mirrors.Left and Vehicle.Body.Mirrors.Right) and it now contains Vehicle.Body.Mirrors.Left.IsHeatingOn and Vehicle.Body.Mirrors.Right.IsHeatingOn signals.

```
IsHeatingOn
  datatype: boolean
  type: sensor
  description: Rear view mirror's heating status
```

Listing 2.1: Mirrors.vspec

```
#include Mirrors.vspec Vehicle.Body.Mirrors.Left
#include Mirrors.vspec Vehicle.Body.Mirrors.Right
```

Listing 2.2: Main.vspec

Overlays provide a way to modify the standard catalog. Consider the Figure 2.2. If we want to add a another signal to it named Vehicle.TestBranch.TestSignal. We specify a new branch as TestBranch and a new signal under TestBranch named TestSignal as shown in Overlay.vspec (see Listing 2.3). Multiple overlays can be added as well. This file is then given to the tool for processing vspec files indicating that this is an overlay.

```
Vehicle:
  type: branch

Vehicle.TestBranch
  type: branch

Vehicle.TestBranch.TestSignal
```

```
    type: actuator
    description: "Test signal"
    datatype: uint32
```

Listing 2.3: Overlay.vspec

2.2.1 Tools and Workflow

[27] is a repository which hosts tools, that convert the VSS tree description given in one or multiple vspec files into Interface Definition Language (IDL) specifications such as Franca [33], Protobuf [34], etc. as well as other formats such as YAML, TTL [68], JSON [17], and GraphQL [35] which can later be used to generate code for interfaces or schema for storing the data internally.

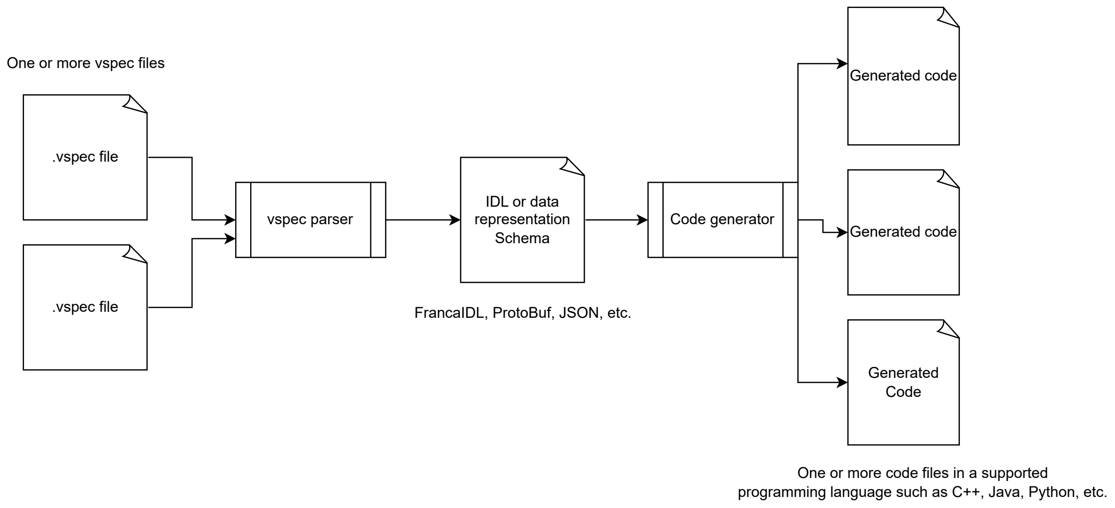

One or more vspec files
.vspec file
.vspec file
vspec parser
IDL or data representation Schema
FrancaIDL, ProtoBuf, JSON, etc.
Code generator
Generated code
Generated code
Generated Code
One or more code files in a supported programming language such as C++, Java, Python, etc.

Figure 2.3: VSS Tools Workflow

2.3 VISS

VISS is an operationalization for accessing VSS signals. For accessing a protected resource, a flow inspired from [37] is used. VISS version 2 defines transport for the following network bindings:

- Secure WebSockets [55]
- Hypertext Transfer Protocol Secure (HTTPS) [63]
- MQTT [60]

Figure 2.4 shows how VISS is related to VSS. There are two specifications, namely Core [13], which describes the general terms and methodology including service interfaces and their functions on a higher level, and Transport [49], which defines the specific implementations with respect to specific network bindings.

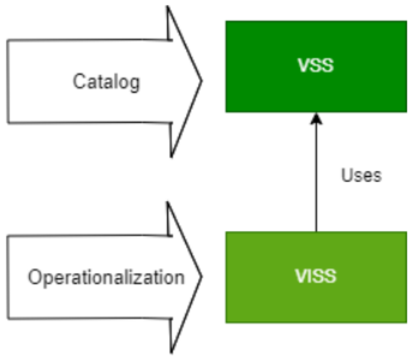


Figure 2.4: Relationship Between VSS and VISS

An implementation for a VISS version 2 server and client is available at [71]. At the time of writing it is implemented in Go programming language.

2.3.1 Methods

Implementing Read and Update methods is mandatory. Subscribe, Unsubscribe and Subscription are optional. Timestamp when used as a field is specified according to [45] as stated in [13]. Only when they are used inside tokens, they must adhere to Unix time. It defines the following methods:

1. **Read:** Used to read one or more signals given a mandatory *path*, an optional *filter* field and an optional *authorization* field in the request. The mandatory *path* field specifies the path of the signal to be read in hierarchical format as specified by VSS. The path can use either '.' or '/' as delimiters. The *filter* field provides additional information to gain more control on what is returned in the response. One can restrict the data retrieved both on a content or temporal basis through the use of filers. Further, read request allows to retrieve some runtime metadata of the VISS services that are utilizing filters. [13] specifies various types of filters in detail. For reading a protected signal, an additional *authorization* field is supplied with a JSON Web Token (JWT) token [47].

   The success response consists of a *timestamp* at which the server executed the request, the *data* which in turn contains the *path* and the *data point* structure which contains the actual *value* of the signal along with a *timestamp* denoting the time of capture of the value. The *value* irrespective of the actual datatype is serialized as a string. The read request and response message structures are depicted in Figure 2.5 and Figure 2.6 respectively.

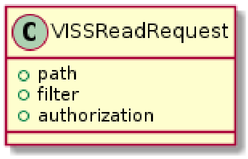


Figure 2.5: VISS Read Request

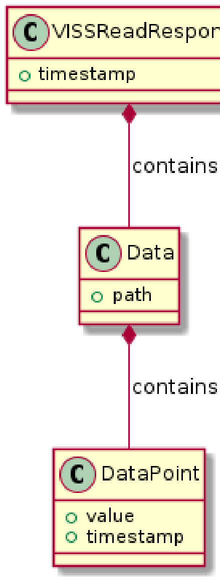


Figure 2.6: VISS Read Response

Listing 2.4 and Listing 2.5 show examples of a Read request and a success response respectively.

```
{
  "path":"Cabin.Door.Row2.Left.IsLocked"
}
```

Listing 2.4: VISS Read Request Example [49]

```
{
  "data": {
    "path":"Cabin.Door.Row2.Left.IsLocked",
    "dp": {"value": "true",
      "ts": "2022-11-09T08:41:00Z"
      }
  },
  "ts": "2022-11-09T08:50:00Z"
}
```

Listing 2.5: VISS Read Success Response Example [49]

2. **Update:** Used to update or write a value to a signal at a given path. For the request *path* and *value* fields are mandatory. For updating a protected signal, a JWT token is provided in an additional *authorization* field. Figure 2.7 depicts the message structure for an update request.

   A success update response contains the *timestamp* of execution of request by the server. Figure 2.8 depicts the structure of success update response.

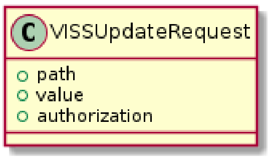


Figure 2.7: VISS Update Request

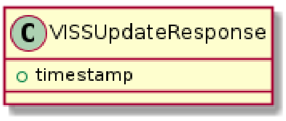


Figure 2.8: VISS Update Response

Listing 2.6 and Listing 2.7 show examples of an Update request and a success response respectively.

```
{
  "path":"Cabin.Door.Row2.Left.IsLocked",
  "value": false
}
```

Listing 2.6: VISS Update Request Example [49]

```
{
  "ts": "2020-04-15T13:37:00Z"
}
```

Listing 2.7: VISS Update Success Response Example [49]

3. **Subscribe:** Used to subscribe to one or more signals given a mandatory *path* and a mandatory *filter* field (if not given, then event issued whenever a new value is published according to [49]). For subscribing to a protected signal, a JWT token is provided in an additional *authorization* field. Figure 2.9 denotes the message structure for a subscribe request.

   The success response contains a *subscriptionId* field which identifies event messages of a particular subscription and *timestamp* denoting the starting time of a subscription. Figure 2.10 denotes the message structure for a subscribe response.

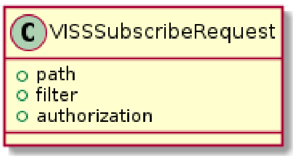


Figure 2.9: VISS Subscribe Request

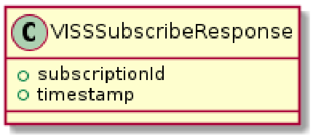


Figure 2.10: VISS Subscribe Response

Listing 2.8 and Listing 2.9 show examples of an Subscribe request and a success response respectively. Filter field is not given as we assume events being published whenever a new value is supplied to the server.

```
{
  "path":"Cabin.Door.Row2.Left.IsLocked"
}
```

Listing 2.8: VISS Subscribe Request Example [49]

```
{
  "subscriptionId": "45678",
  "ts": "2020-04-15T13:37:00Z"
}
```

Listing 2.9: VISS Subscribe Success Response Example [49]

4. **Unsubscribe:** Used to unsubscribe from an existing subscription. The request contains the mandatory *subscriptionId* of a subscription. The success response contains the *subscriptionId* given in the request and a *timestamp* of the time the subscription is terminated. Figure 2.11 denotes the message structure for an unsubscribe request and Figure 2.12 denotes the message structure for an unsubscribe response.

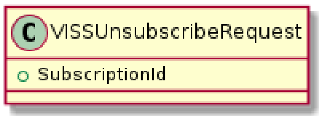


Figure 2.11: VISS Unsubscribe Request

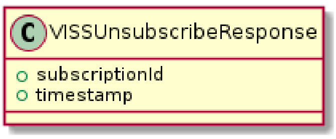


Figure 2.12: VISS Unsubscribe Response

Listing 2.10 and Listing 2.11 show examples of an Unsubscribe request and a success response respectively.

```
{
  "subscriptionId": "45678"
}
```

Listing 2.10: VISS Unsubscribe Request Example [49]

```
{
  "subscriptionId": "45678",
  "ts": "2020-04-15T13:37:00Z"
}
```

Listing 2.11: VISS Unsubscribe Success Response Example [49]

5. **Subscription:** Asynchronous notification message from the server containing the signal, the client has subscribed to. The format is almost the same as a success update response but contains an additional *subscriptionId* identifying the subscription the event message is for. Figure 2.13 denotes the message structure for a subscription message.

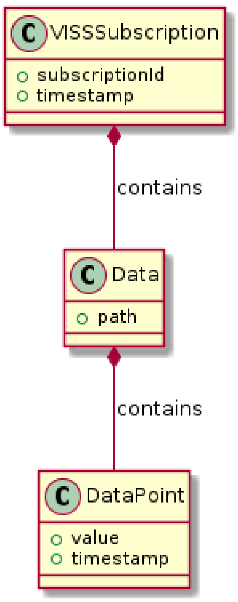


Figure 2.13: VISS Subscription

Listing 2.12 shows an example of a Subscription notification.

```
{
  "subscriptionId": "45678",
  "data": {
    "path":"Cabin.Door.Row2.Left.IsLocked",
    "dp": {"value": "true",
      "ts": "2022-11-09T08:41:00Z"
    }
  },
  "ts": "2022-11-09T08:53:00Z"
}
```

Listing 2.12: VISS Subscription Notification Example [49]

2.3.2 Error Response

In case of an unsuccessful response from server, an error message is used which describes the error. It contains the a *number* identifying an error, a brief *reason* and an associated *message* explaining the error in detail. Apart from these fields, a *timestamp* is provided which tells the time at which the error occurs on the server. A *subscriptionId* field is used in case of on error response to subscribe and unsubscribe request. The *number*, *reason* and *message* are described in *Status Codes* section in [49]. The status codes stated in [58] are to be supported by the client.

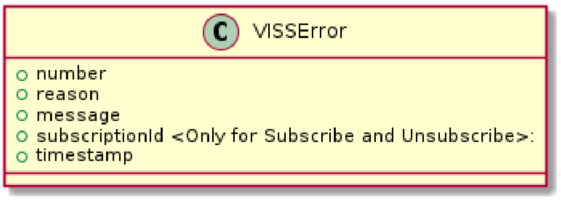


Figure 2.14: VISS Error Response

Listing 2.13 shows an example of an Error response.

```
{
  "error": {
    "number": 406,
    "reason": "not_acceptable",
    "message": "The server is unable to generate content that is acceptable to the client"
  },

  "ts": "2022-11-09T08:53:00Z"
}
```

Listing 2.13: VISS Error Response Example [49]

2.4 SOME/IP

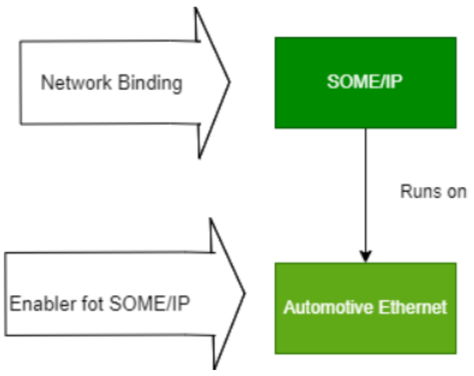


Figure 2.15: Relationship Between SOME/IP and Automotive Ethernet

SOME/IP is an application layer (Open Systems Interconnection (OSI) Layer 7) [16] [15] protocol on the Automotive Ethernet [53] based on Service Oriented Architecture (SOA) principles. It consists of service discovery mechanisms [10], a service oriented protocol layer and a data type system with a serialization scheme [9]. It provides a push and pull based communication by the use of service primitives of methods, fields and events.

A service is identified by the following parameters:

- **Service ID:** It should be a unique ID, identifying each service.
- **Service Instance ID:** When multiple instances of the same service are offered, they can be differentiated using the Service Instance ID.
- **Major Version:** Used to denote the offered service instance's major version.
- **Minor Version:** Used to denote the offered service instance's minor version.

2.4.1 Message Format

Figure 2.16 shows how the SOME/IP message with header and payload looks like. Fields from **Message ID** till **Return Code** constitute the SOME/IP header and the rest is payload. The important fields are described below.

- **Message ID:** This 32 bit field is made up of 16 bits of Service ID (Most Significant Byte (MSB)) and 16 bits of Method ID (Least Significant Byte (LSB)). The Method ID differs for each methods (0x0000 to 0x7fff) and events (0x8000 to 0x8fff) of a service.
- **Length:** This 32 bit field denotes the length in bytes including Request ID to all the way up to the end of the message payload.
- **Request ID:** This field allows both client and server to uniquely identify requests of the same method. This 32 bit field consists of 16 bits client ID (MSB) and 16 bits session ID (LSB).
- **Protocol Version:** 8 bits denoting the version of SOME/IP header format being used.
- **Interface Version:** 8 bits denoting the service interface version in use.
- **Message Type:** 8 bits denoting SOME/IP message's type. Some of the relevant message types are described in Table 2.1. For more message types refer to Table 4.4 in [9].
- **Return Code:** 8 bits denoting success if 0x00 and if anything else, denotes error. It is used in requests as well but set to 0x00. Some relevant return codes are described in [9].
- **Payload:** This contains the actual content or as the name suggests, the payload of a SOME/IP message.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Message ID (Service ID / Method ID) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Length [32 bit]                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (Client ID / Session ID) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|
Protocol Vers.| Interface Ver.| Message Type .| Return Code     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                     Payload [variable size]                   +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.16: SOME/IP Message Format [9]

Message Type	**Meaning**
0x00	Request Message expecting response
0x01	Request Message expecting no response
0x80	Success Response Message
0x81	Error Response Message
0x02	Request for Notification expecting no response

Table 2.1: Message Types [9]

Message Type	**Allowed Return** Codes
0x00	0x00 (E_OK)
0x01	0x00 (E_OK)
0x02	0x00 (E_OK)
0x80	Refer to Table 4.11 in [9]
0x81	Refer to Table 4.11 in [9] (anything except 0x00)

Table 2.2: Allowed Return Codes

2.4.2 Service Discovery

SOME/IP-SD [10] makes the discovery of services by the clients on runtime as well as subscription mechanisms possible. The constraint is that SOME/IP-SD messages are only to be sent over User Datagram Protocol (UDP). The discovery of services happen over a multicast group (in case of dynamic service discovery, otherwise no need of service discovery if static configuration is used). We first describe the message format used, then the accompanying entries and options and finally the flow of service discovery at the end. This standard also specifies event and field subscription semantics. Figure 2.17 shows the structure of a SOME/IP-SD message. SOME/IP-SD messages set some values of the generic SOME/IP Protocol header fixed per SOME/IP-SD standard. The **Service ID** is set to **0xFFFF** and the **Method ID** to **0x8100**. The **Message Type** field is set to **0x02**. Both the **Protocol Version** and **Interface Version** fields shall be set to **0x01**.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                        SOME/IP Header                         +
|                        [128 bits]                             |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Flags [8 bit]|               Reserved [24 bit]               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               Length of Entries Array [32 bit]                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                         Entries Array                         +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               Length of Options Array [32 bit]                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                         Options Array                         +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.17: SOME/IP-SD Message Format [10]

Entries and Options

The SOME/IP-SD payload contains an array of entries and an array of options. The entries indicate the intent of the client or server on SOME/IP-SD level and it is divided into two types:

- **Service entry:** indicates entries of types 0x00 (FindService), 0x01 (OfferService) and 0x01 (StopOfferService). The OfferService entry (Figure 2.18) is used to announce the service instance by a server offering that specific service. StopOfferService entry (Figure 2.19) is used indicate that a service offered by a service instance is not valid anymore by setting the Time To Live (TTL) field to 0. In a service entry, one can reference options present in the options array from 2 range of indexes given by Ind_1 and Ind_2 respectively and the number of options referenced from both ranges are indicated by n1 and n2 fields respectively. In this case, one option is referenced from index 0 indicated by n1 and Ind_1 respectively. For range starting from index Ind_2, no options are referenced as n2 is 0. The service ID is 0x1000, Instance ID is 0x0001 and the Major and Minor version fields are 0x01 and 0 respectively. The TTL of 3 indicates that this offer is valid for 3 seconds. FindService entry is used to find a service instance manually Figure 2.20. A specific Instance ID can be given or 0xffff if any Instance ID of the service is sufficient.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Type (=0x01) |  Ind_1 (=0x00)|  Ind_2 (=0x00)| n1 (1)| n2 (0)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Service ID (=0x1000)     |      Instance ID (=0x0001)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Major (=0x01)|              TTL (=0x000003)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Minor Version (=0x00000000)                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.18: SOME/IP-SD OfferService Entry Example [10]

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Type (=0x01) |  Ind_1 (=0x00)|  Ind_2 (=0x00)| n1 (1)| n2 (0)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Service ID (=0x1000)     |      Instance ID (=0x0001)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Major (=0x01)|              TTL (=0x000000)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Minor Version (=0x00000000)                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

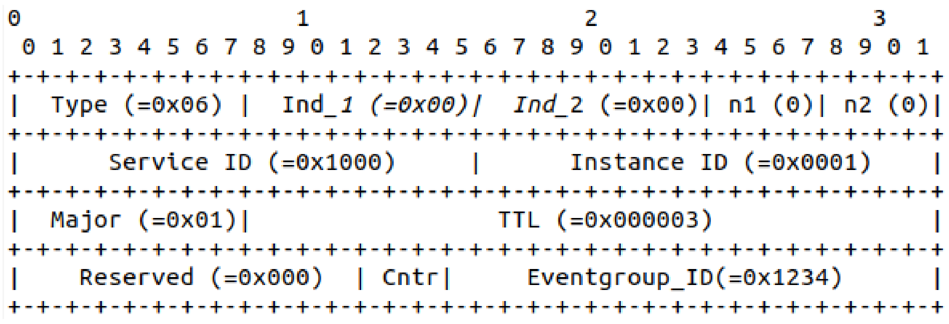
Figure 2.19: SOME/IP-SD StopOfferService Entry Example [10]

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Type (=0x01) |  Ind_1 (=0x00)|  Ind_2 (=0x00)| n1 (0)| n2 (0)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Service ID (=0x1000)     |      Instance ID (=0xffff)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Major (=0x01)|              TTL (=0x000000)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Minor Version (=0x00000000)                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.20: SOME/IP-SD FindService Entry Example [10]

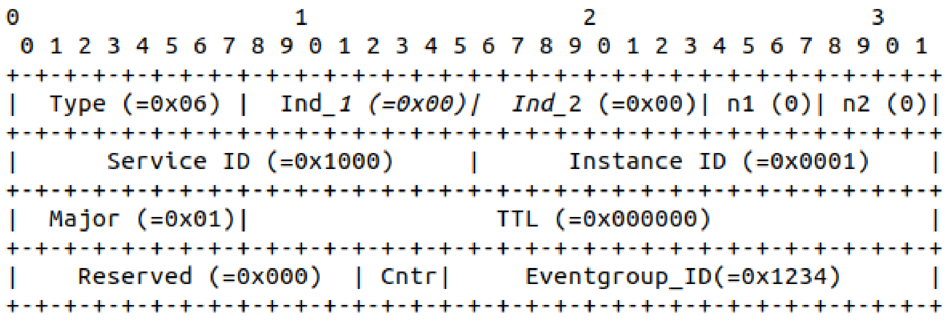
- **Eventgroup entry:** indicates entries of types 0x06 (Subscribe), 0x06 (StopSubscribeEventgroup), 0x07 (SubscribeAck) and 0x07 (SubscribeEventgroupNack). These entries are similar to Service entries but instead of Minor version field have a Reserved field (which is always 0), a counter field (Cntr), which can be increment in case of parallel subscribes to the same eventgroup by the same client and an Eventgroup ID.

  Figure 2.21 and Figure 2.22 show examples of Subscribe and StopSubscribe entries for subscribing and unsubscribing to eventgroup with ID 0x1234 which have the same type (0x06) with the difference being that the Subscribe entry has a TTL equal to the amount of time in seconds for which the subscription is valid (In this case 3 seconds), but StopSubscribe has 0.



```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Type (=0x06) |  Ind_1 (=0x00)|  Ind_2 (=0x00)| n1 (0)| n2 (0)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Service ID (=0x1000)     |      Instance ID (=0x0001)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Major (=0x01)|                 TTL (=0x000003)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Reserved (=0x000)  | Cntr|     Eventgroup_ID(=0x1234)      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

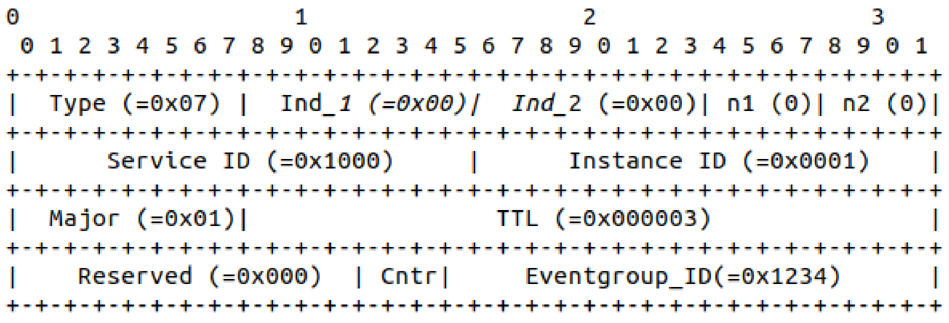
Figure 2.21: SOME/IP-SD Subscribe Entry Example [10]



```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Type (=0x06) |  Ind_1 (=0x00)|  Ind_2 (=0x00)| n1 (0)| n2 (0)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Service ID (=0x1000)     |      Instance ID (=0x0001)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Major (=0x01)|                 TTL (=0x000000)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Reserved (=0x000)  | Cntr|     Eventgroup_ID(=0x1234)      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.22: SOME/IP-SD StopSubscribe Entry Example [10]

Figure 2.23 and Figure 2.24 show examples for positive subscribe acknowledgment (SubscribeAck) and negative subscribe acknowledgment (SubscribeNack) for a Subscribe entry which is shown in Figure 2.21. The SubscribeAck and SubscribeNack have the same type (0x07), but SubscribeAck copies the TTL from the Subscribe entry sent by a client, whereas the SubscribeNack makes it 0 and is sent in case the subscription is unsuccessful.



```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Type (=0x07) |  Ind_1 (=0x00)|  Ind_2 (=0x00)| n1 (0)| n2 (0)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Service ID (=0x1000)     |      Instance ID (=0x0001)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Major (=0x01)|                 TTL (=0x000003)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Reserved (=0x000)  | Cntr|     Eventgroup_ID(=0x1234)      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.23: SOME/IP-SD SubscribeAck Entry Example [10]

```
 0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |  Type (=0x07) |  Ind_1 (=0x00)|  Ind_2 (=0x00)| n1 (0)| n2 (0)|
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |      Service ID (=0x1000)     |       Instance ID (=0x0001)    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |  Major (=0x01)|                TTL (=0x000000)                 |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |    Reserved (=0x000)   | Cntr|     Eventgroup_ID(=0x1234)      |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.24: SOME/IP-SD SubscribeNack Entry Example [10]

The entries then refer to one or more options present in the array of options. The options are used supply additional information to supplement entries like endpoints, additional configurations, etc. The following types of options are available under SOME/IP-SD:

- **Configuration:** Provides additional configuration in form of key value pairs provided in a configuration string. This string is an array of American Standard Code for Information Interchange (ASCII) characters, where a key value pair starts with the total the length of key, value and and equals character '=' and repeats the same for multiple keys until the value 0 is reached indicating the termination of configuration string. Figure 2.25 shows the format for such an option. The Type field is set to 0x01 and the Reserved field is set to 0. D stands for discardable field which is exactly 1 bit.

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |             Length            |   Type(0x01)  |D|   Reserved  |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                                                               |
 +     Zero-Terminated Configuration String ([len]id=value[0])   +
 |                                                               |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.25: Configuration Option [10]

- **Load Balancing:** Provides information required by the client to prioritize service instances based on priority and weight criteria similar to DNS-SRV [36] records. The type field is set to 0x02 in this case.
- **IPv4 Endpoint:** Provides information on possible IPv4 unicast address and the transport layer protocol used by a client or a server instance. The type field is set to 0x04 in this case.
- **IPv6 Endpoint:** Provides information on possible IPv6 unicast address and the transport layer protocol used by a client or a server instance. The type field is set to 0x06 in this case.
- **IPv4 Multicast:** Provides information on possible IPv4 multicast address and the transport layer protocol used by a client or a server instance. The type field is set to 0x14 in this case.
- **IPv6 Multicast:** Provides information on possible IPv6 multicast address and the transport layer protocol used by a client or a server instance. The type field is set to 0x16 in this case.

- **IPv4 Service Discovery (SD) Endpoint:** Provides information on possible IPv4 SD address of the sender and the transport layer protocol used by it. The type field is set to 0x24 in this case.
- **IPv6 SD Endpoint:** Provides information on possible IPv6 SD address of the sender and the transport layer protocol used by it. The type field is set to 0x26 in this case.

Message Flow

Figure 2.26 shows the flow for the Service Discovery phase. The server sends cyclic SOME/IP-SD messages with an OfferService entry over a preconfigured multicast address over UDP in case of dynamic service discovery. In case of static configuration, clients already know the endpoint and hence no service discovery messages are sent. The client can listen for these messages or can do an on-demand lookup for services by sending SOME/IP-SD message with a FindService entry, and in return it receives the OfferService entry from the server.

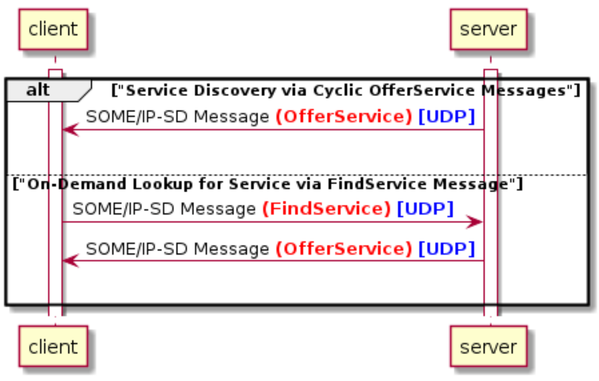

client
server
alt
["Service Discovery via Cyclic OfferService Messages"]
SOME/IP-SD Message (OfferService) [UDP]
["On-Demand Lookup for Service via FindService Message"]
SOME/IP-SD Message (FindService) [UDP]
SOME/IP-SD Message (OfferService) [UDP]
client
server

Figure 2.26: Service Discovery Flow [10]

2.4.3 Methods

These are remote procedures that a client can invoke. A server may or may not respond depending on whether it is a request-response or a fire and forget method type respectively. Figure 2.27 shows an example flow for both types of method calls.

Figure 2.28 shows structure of a request message. The Message Type is always set to 0x00 and return code to 0x00 as well.

Figure 2.29 shows the structure of a success response message. The Message ID and Request ID fields are the same as the request, but the Message Type field is set to 0x80 and the Return Code to 0x00.

Figure 2.30 shows the structure of an error response message. The Message ID and Request ID fields are the same as the request, but the Message Type field is set to 0x81 and the Return Code to one of the error codes as described in Table 4.11 in [9].

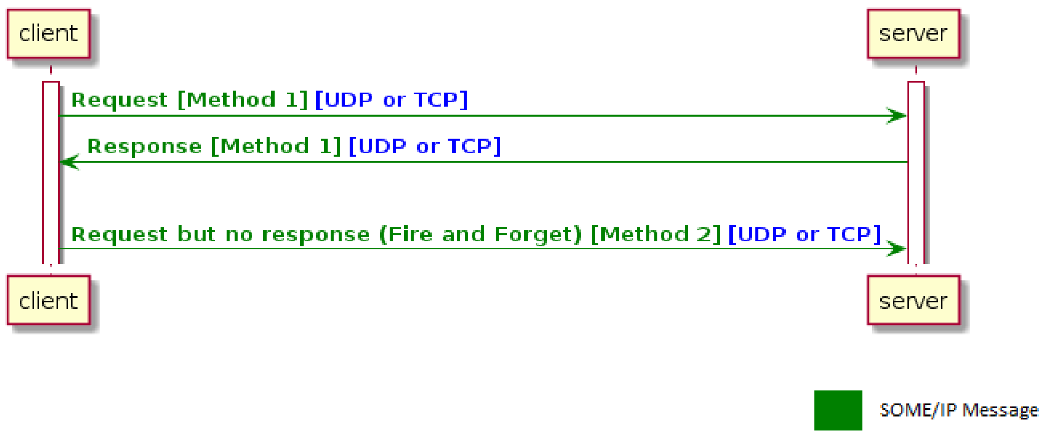

client
server
Request [Method 1] [UDP or TCP]
Response [Method 1] [UDP or TCP]
Request but no response (Fire and Forget) [Method 2] [UDP or TCP]
client
server
SOME/IP Message

Figure 2.27: Methods Flow [9]

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Message ID [32 bit]                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Length [32 bit]                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (Client ID / Session ID) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ = 0x00 | RetCode = 0x00|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                    SOME/IP Request Payload                    +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.28: SOME/IP Request Message

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Message ID [32 bit]                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length [32 bit]                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (Client ID / Session ID) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ = 0x80 | RetCode = 0x00|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                     SOME/IP Response Payload                  +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.29: SOME/IP Success Response Message [9]

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Message ID [32 bit]                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length [32 bit]                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (Client ID / Session ID) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ = 0x81 | Error Code    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                    SOME/IP Error Payload                      |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.30: SOME/IP Error Response Message [9]

2.4.4 Events

These are data which need to be transmitted by the server and all the subscribing clients receive the data along with the event notification for the same. Figure 2.31 shows an example flow for subscribing and unsubscribing event groups. One point to note is that, a client can only subscribe to a whole event group and not an individual event.

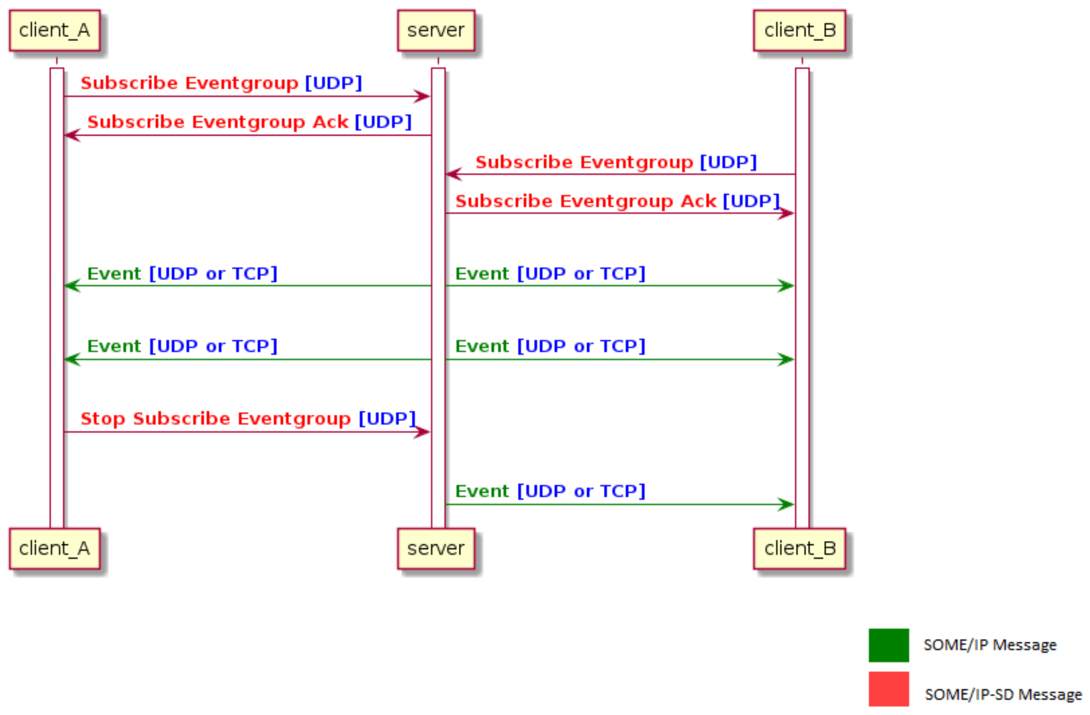


Figure 2.31: Events Flow [9]

The event notification message structure is shown in Figure 2.32. The Method ID can be set to any value but recommended to be in the range 0x8000 to 0x8fff to be able to differentiate between requests / responses and event notifications. In the Request ID field, the Client ID is always set to 0x00, but the session ID can be set depending on whether session handling is active or inactive (inactive then 0x00, else increment between the range 0x1 to 0xffff depending on the use case). The Message Type in this case is always 0x02.

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Message ID (Service ID / Method ID) [32 bit]             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Length [32 bit]                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (0x00 / Session ID) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.|  Msg_typ=0x02 / Ret_code (0x00)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                      Payload [variable size]                  +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.32: SOME/IP Event Notification Message [9]

2.4.5 Fields

These are a combination of methods and events. These are data consisting of at least one of the three: a getter, a setter or a notifier. In case a notifier is present, the field transmits the value as an event to all it's subscribers whenever it is changed. The getter is a request/response method without any value in the request and the setter has the value in the request. Field notifications use the same message format as event notifications.

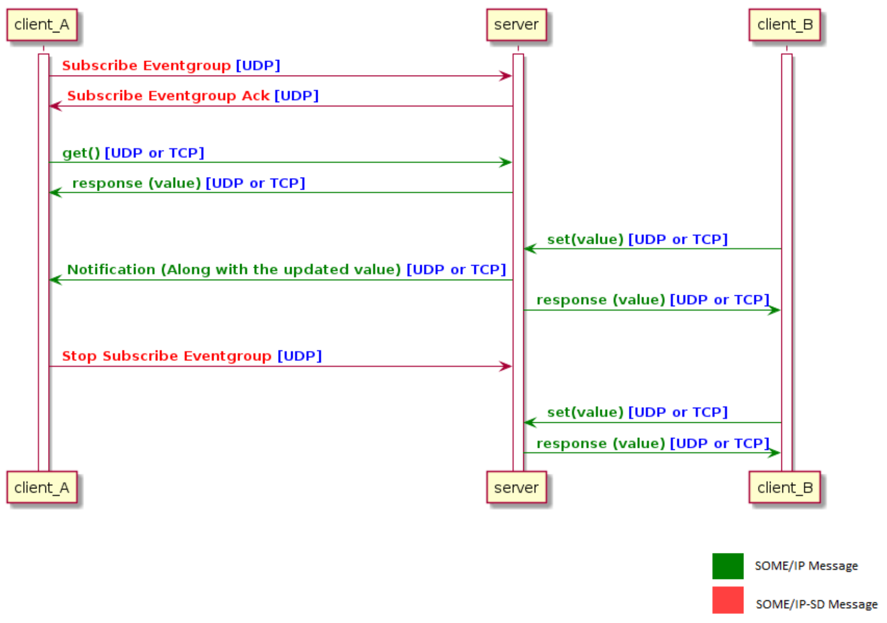

client_A
server
client_B
Subscribe Eventgroup [UDP]
Subscribe Eventgroup Ack [UDP]
get() [UDP or TCP]
response (value) [UDP or TCP]
set(value) [UDP or TCP]
Notification (Along with the updated value) [UDP or TCP]
response (value) [UDP or TCP]
Stop Subscribe Eventgroup [UDP]
set(value) [UDP or TCP]
response (value) [UDP or TCP]
client_A
server
client_B
🟩 SOME/IP Message
🟥 SOME/IP-SD Message

Figure 2.33: Fields Flow [9]

2.4.6 Serialization

SOME/IP protocol [9] states the general communication protocol consisting of details about header and payload structure, serialization of and other aspects like request response functionality. According to the standard, basic data types supported are:

- Boolean [1 Byte]
- Floating Point Number [4 / 8 Bytes]
- Unsigned Integer [1 / 2 / 4 / 8 Bytes]
- Signed Integer [Same sizes as Unsigned Integer]

Apart from the standard types, support for more complex data types is also available. These include:

- String (fixed and dynamic length)
- Array (fixed and dynamic length)
- Multidimensional array
- Structured Datatype (struct)
- Union /variant
- Enumeration
- Bitfield

TLV

SOME/IP standard also specifies a TLV based serialization scheme. The members of a struct can be attached with a tag consisting of Reserved Bit, Wire Type and Data ID to allow serialization of out of order and optional members. Figure 2.34 and Figure 2.35 show the layout for a such a tagged member with static and dynamic lengths. R stands for Reserved Bit, Wty for Wire Type (3 bits). The Data ID is 12 bits long followed by an optional length field depending on the Wire Type which can be 8, 16 or 32 bits long and then finally the member.

Table 2.3 shows the wire type values and their corresponding data types. For wire types 0 to 4, the length is preconfigured and hence is not needed separately. For types 5 to 7 the length field is required and can be 1 byte to 4 byte depending on the type as described in Table 2.3.

Wire Type	**Data Type**
0 - 3	8, 16, 32 and 64 bit data
4	Complex Data Type (Static Size)
5	Complex Data Type (Length Field Size = 1 Byte)
6	Complex Data Type (Length Field Size = 2 Byte)
7	Complex Data Type (Length Field Size = 4 Byte)

Table 2.3: Wire Types and their Corresponding Data Types [9]

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|R| Wtyp|         Data ID       |           Member...           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.34: TLV Serialization of Data Type without Length Field [9]

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|R| Wtyp|         Data ID       |     Length    |    Member...  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.35: TLV Serialization of Data Type with Length Field [9]

Union / Variant

SOME/IP offers the possibility to serialize unions / variants as well. This works by adding additional metadata before the union. This metadata consists of Length field (8, 16, 32 or 0 bits) and a Type field (8, 16 or 32 bits). The bit length for both the fields can be defined in the configuration. The Type ID for each supported datatype has to be defined in the configuration (except 0 as it is reserved for NULL type). Figure 2.36 shows the format for serialization of unions using the metadata.

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 Length Field [0/8/16/32 bits]                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Type field [8/16/32 bits]                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Data including padding                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.36: SOME/IP Union / Variant Serialization with Metadata [9]

3 State of the Art

The term Vehicle API is used to define APIs ranging from those which connect a vehicle to the cloud and to those which extract information from a vehicle, without it having to be directly involved (like scanning VIN and license plates of a vehicle, see [6]).

When it comes to Vehicle API, SOVD [66] fulfills a similar purpose, but extends beyond just accessing and modifying vehicle signals as it provides vehicle specific diagnostic functionalities which also include updating the software within the ECUs. Another similar concept is VSC [23], but it can be used to define service interfaces having methods, events and properties as well. It acts like an IDL capable of defining Original Equipment Manufacturer (OEM) and vehicle model specific complex data types and service interfaces. It does not aim to provide a standardized catalog of signals like VSS but provides tools to make a vehicle model specific service catalog [5].

According to [52], a majority of these features are possible due to internet access in today's vehicles, especially the Built-in Connectivity. Vehicle Telematics has been one area where such technologies have been in use, for example in motor insurance based on the usage criteria [38]. This data collection can help the OEM improve the existing vehicle by analyzing the vast amount of driving data, and getting to know the root cause of incidents in more detail and provide a better product and services to it's customers. [48] states some relevant standards and existing implementations for Vehicle APIs concerning access to data generated by the vehicle. Standards include ISO 20077 [40] [41], ISO 20078 [42] [43] for Extended Vehicle (ExVe) methodology, and ISO 20080 [44] for remote diagnostics.

API implementations include BMW CarData [14] based on the Neutral Extended Vehicle for Advanced Data Access (NEVADA) [48] concept, now succeeded by the Automotive Data Access - Extended and Open (ADAXO) [70] concept by Verband der Automobilindustrie (VDA). Mercedes Benz also provides such APIs on it's Meredes-Benz /developers [56] portal.

[26] lists up to date information on new VSS implementations and it's current usage by the OEMs. At the time of writing the following open source implementations of VSS are listed on this site:

- playground.digital.auto [28]
- Aos Edge [32]
- Eclipse Kuksa [67]
- AWS IoT Fleetwise [2]
- VISS by W3C [71]

According to [72] the use of APIs inside a vehicle is mostly visible within the In-Vehicle Infotainment (IVI) units, allowing them to access 3rd party services on the internet. Another use is to expose certain functionality of the vehicle to allow developers to use them while developing applications for these specific IVI units.

Android Automotive OS introduced the concept of Vehicle Hardware Abstraction Layer (VHAL) [3]. It provides vehicle signals as vehicle properties [4]. OEMs need to then implement ways to access these signals. These can be implemented in multiple ways. For example, via bus systems like Controller Area Network (CAN) or via SOME/IP over Automotive Ethernet. It is theoretically possible to use VSS as well as long as these properties are accessible via VISS or any other VSS implementation as vehicle signals over a suitable transport and interfaces required by the VHAL are implemented.

In terms of wired networking inside a modern vehicle, according to [53], CAN, Local Interconnect Network (LIN), Media Oriented System Transport (MOST), FlexRay, Ethernet and a few other technologies are in use. But Ethernet is increasingly finding it's use in many onboard applications due to it's higher bandwidth and a more general purpose characteristics.

According to [64], SOA is finding its usage inside vehicles due to it's dynamic reconfigurability which allows for greater flexibility in terms of upgrading onboard software. Application layer protocols namely SOME/IP and DDS are specified in newer software standards used in the automotive industry, for example, AUTomotive Open System ARchitecture (AUTOSAR) Adaptive [8].

[54] provides a comparison between MQTT, SOME/IP and DDS. It also states that currently SOME/IP and DDS have official support in AUTOSAR, whereas MQTT does not. But in terms of adaptation in AUTOSAR for in-vehicle communication, according to [59] and [51], SOME/IP has currently an upper edge as compared to DDS and MQTT.

In terms of Inter Process Communication (IPC), automotive middlewares are taking zero-copy and shared memory approaches like Eclipse Iceoryx [30].

4 Design and Analysis

4.1 Some questions regarding SOME/IP as transport binding

While considering SOME/IP as an alternative transport binding, we need to answer a few questions which will influence the design of the overall solution, these include:

- How do we design the SOME/IP service? How can the Service Interfaces be versioned?
- Can the VISS based subscriptions be mapped directly on SOME/IP?
- Do we use JSON for packing requests and responses on the SOME/IP side or do we take advantage of standard data types and serialization options provided by SOME/IP? What are the advantages and disadvantages of using these approaches?
- What approaches can be used for address resolution? For example, there can be data points for front left and front right doors whose addresses are **Vehicle.Chassis.Front.Left.Door** and **Vehicle.Chassis.Front.Right.Door**. Do we store both values on a centralized server or take advantage of the distributed nature of ECUs within a vehicle?
- What about the security aspects?

4.2 Service Design

There are three major approaches, with the help of which the services can be designed for our use case, these are listed below:

4.2.1 Single Method for all VISS Operations

In this approach, a single SOME/IP method is used for all possible VISS operations. In this scenario a VISS request is passed to the server to a method with a predefined method id, for example 0x0001, and it is the responsibility of the server to check the request contents to determine which type of operation (VISS Read, Update, etc.) corresponds to it and process it accordingly.

Advantages

1. Service interface is simpler as there is only one method.
2. Signals can be distributed between ECUs as only the method signature need to be the same between them.
3. Read and subscribe requests with filters can implemented as a part of the single method skeleton on the server side as well, but subscription handling has to be taken care in a dedicated way for each subscription on the server side.

Disadvantages

1. Logic of the skeleton can become complex as all possible VISS requests and responses have to be processed in a single method.

4.2.2 Separate Method for each VISS Operation

In this approach, each VISS operation is mapped to its own SOME/IP method. This means that the client can just call the specific operation separately for each VISS operation. For example method id 0x0001 for VISS Read, 0x0002 for VISS Update and so on.

Advantages

1. Logic of the skeleton can be implemented flexibly as it is divided into multiple methods.
2. Signals can be distributed between ECUs as only the method signatures need to be the same between them.
3. Read and subscribe requests with filters can implemented as method skeletons on the server side as well, but subscription handling has to be taken care in a dedicated way for each subscription on the server side.

Disadvantages

1. Service Interface is more complex than the previous approach due to multiple methods.

4.2.3 Signals Modeled as Fields

SOME/IP provides fields which allows data to be subscribed to and have getters and setters. Each signal can be made a field with it's dedicated eventgroup so that each signal's subscription is separate from the other. Getters would act as read method, setters as update method.

Advantages

1. Focus is only on data, and rest is automatically taken care by the semantics of fields.

Disadvantages

1. Good for read and subscribe requests without filters. Read and subscribe requests with filters cannot be implemented directly with field's subscription logic.
2. Cannot map the VSS signal tree partially between different service instances as all the fields need to be implemented by a given service instance. This is due to the fact that since signals are now fields, they become a static part of a given service interface.

4.3 Service Interface Versioning

While versioning the service interface for the service, the major version is incremented when big changes (or in other terms breaking changes) are introduced. The minor version is incremented when new functionality is added without breaking compatibility with the older version. For example, addition of a new field or method, but keeping the old ones as it is, is not a breaking change, but modifying the signature of a preexisting function or a field would be a breaking change.

In our scenario, VISS and VSS have separate versioning scheme, so it is important to decide how the major and minor versions of our service interface will be affected. As of writing the report, VSS is at version 2 and draft date of 18th April 2023. While VSS is at version 4.

[13] does not specifically state any particular version of VSS, but VSS could also introduce breaking changes atleast in the semantics of data (removal or addition of new data types or modification to current datatypes). For example, VSS version 4 introduced support for structs.

In order to align both standards, one needs to somehow decide upon two major approaches:

1. **Use own arbitrary versioning scheme** to which the exact VSS and VISS versions are mapped. For example, Major Version 0x01, Minor Version 0x00000002 could mean that it uses version 2 of VISS, draft date 18th April 2023 and VSS version 4. The only downside is that one then needs to keep track of all this information.
2. **Combine both versioning schemes** such that the developer and / or clients and servers can decode and encode the versioning information themselves. One has to keep in mind that SOME/IP defines major version field (or interface version) to be only 8 bits. Whereas minor version to be 32 bits. Which means that the major version can contain the major version of either VISS or VSS. And the minor version can be a combination of minor version of one and major and minor version of the other. Since we are describing ways of mapping VISS to SOME/IP, we can consider major version field to contain the major version of VISS and the minor version to contain a combination of minor version of VISS and major and minor versions of VSS. For example Major Version 0x02 and Minor Version 0x0AC10AE8. The Major Version represents version 2 of VISS and the minor version when interpreted in decimal is 180423400 which can be interpreted as 18.04.23 which is the draft date of VISS

version (18th April 2023) and the remaining 3 digits of the decimal representation 400 as 4.0.0 or version 4 of VSS. This is just an example and not a rule to be followed. One can choose own convention, but it should be fixed so that only the convention needs tobe known and not the version details (as they can be extracted from it).

4.4 Request ID Generation

An approach suggested in [9], states that the client IDs can be fixed such that they remain unique among each client. Then a random session ID can be generated by each client such that the same session ID does not repeat for the same client at the same time. This should work for our design, because the clients are generally known at design time. As long as two clients do not share the same client ID, the request ID will always be unique. If multiple applications are running on the same ECU, we suggest:

1. If the applications are known at design time and do not change, then each app can have one or multiple fixed client IDs depending on the need.
2. If the applications are not known at the design time like in IVI ECUs, then there can either be a single client per ECU like a daemon with a unique client ID which processes the VISS requests for each application. Or Each ECU can have a pool of client IDs which are mutually exclusive from the ones in another ECU and are allocated to each application on need basis and released when not required by that application, so that they can be used by another application. These client IDs would then be required to kept track of by each ECU itself.

4.5 Handling Subscriptions

Read and update request / response can be directly mapped to SOME/IP methods. But when it comes to subscription, it gets complicated due to the fact that SOME/IP-SD standard manages the subscriptions via eventgroups, which means that a client subscribes not just to a single event, but rather to an eventgroup. The issue can be resolved to an extent if we assign each signal to it's own eventgroup, but we still cannot manage subscriptions with filters which might be different for different clients. Hence, we need a way to manage subscriptions independent of the SOME/IP-SD standard. Since, event notifications are sent via SOME/IP standard just like requests and responses. It means, we can implement subscribe and unsubscribe as methods and subscriptions can then be managed on server (to send notifications depending on the condition specified by the client in request) as well as client side (so that the client keeps the individual connection alive or endpoint open as long as it is subscribed). This might mean that we loose some of the optimizations like grouping of event notifications on multicast addresses, but we gain more flexibility required for a data-oriented approach.

Server can keep a key value pair, where the key is the signal name, and the value is a list of (subscriptionId, client connection, rule_function). Client connection is the information required to send notification to remote client and rule function is the function which can check if a condition as specified by the client in filter is satisfied and if satisfied then send. The server can store this information before sending the subscribe positive reply. A publish function on the server can be made which accesses the key value pair and checks the condition for each remote client of a particular signal and send notification accordingly to all subscribers satisfying the criteria.

The client upon receiving a subscribe success response can store the information in it's own local key value pair, where key is the subscriptionId and value is the information required to connect to the remote server.

In case of an unsubscribe, the client can check the remote server info in the key value pair for a specific subscriptionId and send the request accordingly. This way on success, the server and client can remove the entry in their respective key value pairs.

4.6 Serialization of VISS Requests and Responses

There are multiple possibilities for modeling requests and responses as long as the request and response data structures use SOME/IP supported data types and serialization as described in [9].

4.6.1 Using JSON

In this approach, we wrap the VISS JSON [17] requests and responses (as described in [49]) in SOME/IP request and response message payloads. This is done by using the string data type (Unicode Transformation Format-8 (UTF-8) [73]). Note that all strings would include 3 bytes Byte Order Mark (BOM) and 1 byte termination character (here 0). So each string length would be 4 bytes plus it's original length.

Figure 4.1 shows the VISS request payload. The Message ID is set to the respective Service ID and the Method ID to tell which VISS operation request is being referred to. Message Type is set to 0x00 and Return Code to 0x00 as well. The JSON encoded VISS request can be serialized in the VISS Request String field as a UTF-8 dynamic string and it's length is indicated by the VISS Request Length field in SOME/IP payload. Figure 4.2 shows an example of such a request with client ID to be 1001.

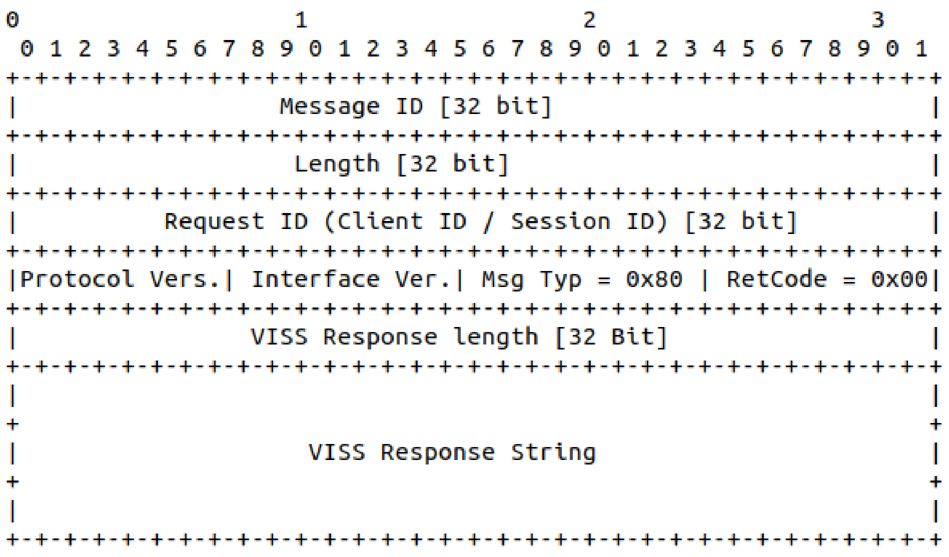
```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Message ID [32 bit]                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length [32 bit]                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (Client ID / Session ID) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ = 0x00 | RetCode = 0x00|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                VISS Request length [32 Bit]                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                    VISS Request String                        |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.1: VISS Request using JSON Serialization

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0001) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length  (0x0000 0038) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (0x1001 1234) [32 bit]                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ (0x00) | RetCode (0x00)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Request_length (32 Bit) : 0x0000 002C          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
               Request (44 Bytes):
       {
              "path":"Cabin.Door.Row2.Left.IsLocked"
       }

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

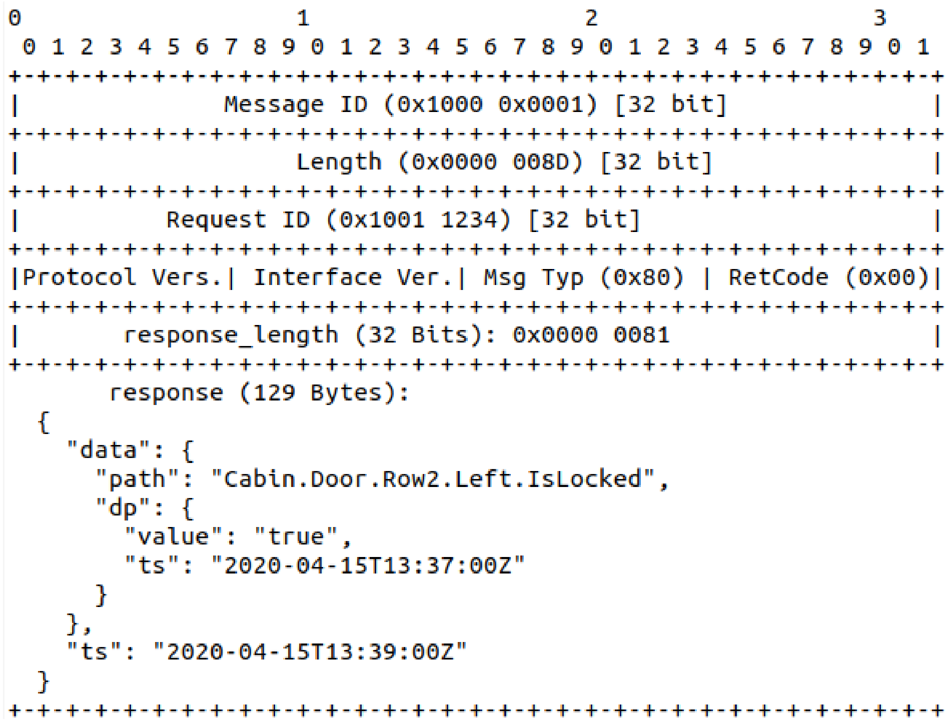
Figure 4.2: Example VISS Request using JSON Serialization

Figure 4.3 shows the VISS success response payload. The Message ID is set to the respective Service ID and the Method ID to tell which VISS operation response is being referred to. The Message Type is set to 0x80 and the Return code to 0x00. Rest is same as request payload, but instead of request, we serialize the success response in JSON as a UTF-8 dynamic string with the help of VISS Response Length and VISS Response String fields. Figure 4.4 shows an example success response to the request shown in Figure 4.2.



```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Message ID [32 bit]                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Length [32 bit]                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (Client ID / Session ID) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ = 0x80 | RetCode = 0x00|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 VISS Response length [32 Bit]                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                     VISS Response String                      |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.3: VISS Success Response using JSON Serialization



```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0001) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length (0x0000 008D) [32 bit]               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (0x1001 1234) [32 bit]                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ (0x80) | RetCode (0x00)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       response_length (32 Bits): 0x0000 0081                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       response (129 Bytes):
  {
    "data": {
      "path": "Cabin.Door.Row2.Left.IsLocked",
      "dp": {
        "value": "true",
        "ts": "2020-04-15T13:37:00Z"
      }
    },
    "ts": "2020-04-15T13:39:00Z"
  }
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.4: Example VISS Success Response using JSON Serialization

Figure 4.5 shows a possible structure for a VISS error response payload. The Message ID field has the same function as in VISS request and success response as shown earlier. We can use the built-in mechanism of SOME/IP Message Type 0x81 and error code E_NOT_OK by using Return Code

0x01 in the response. we serialize the error response in JSON as a UTF-8 dynamic string with the help of VISS Error Length and VISS Error String fields. 4.6 shows an error response to the request shown in Figure 4.2.

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Message ID [32 bit]                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length [32 bit]                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (Client ID / Session ID) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ = 0x81 | RetCode = 0x01|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 VISS Error length [32 Bit]                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                    VISS Error String                          |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.5: VISS Error using JSON Serialization

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0001) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length (0x0000 0094) [32 Bit]               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (0x1001 1234) [32 bit]                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ (0x81) | RetCode (0x01)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Error_length (32 Bit): 0x0000 0084                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        Error (132 Bytes):
        {
        "error": {
           "number": 404,
           "reason": "invalid_path",
           "message": "The specified data path does not exist."
           },
          "ts": "2020-04-15T13:37:00Z"
        }
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.6: Example VISS Error Response using JSON Serialization

Figure 4.7 shows the possible structure for a VISS subscription payload. One can choose to be in the range of 0x8000 to 0x8fff as suggested by SOME/IP standard. But we are going a bit beyond the standard, hence 0x0005 as method ID. The request ID is always 0, session ID could be incremented if session handling is used. But here we use 0 as we are not doing session handling for subscriptions.

The message type is 0x02 (notification) and return code of 0. The actual JSON payload is contained in VISS subscription string field, whose length is indicated by a 32 bit VISS Subscription length field. Figure 4.8 shows an example of a VISS subscription.

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Message ID (Service ID / 0x0005) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length [32 bit]                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (0x0000 0000) [32 bit]                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ = 0x02 | RetCode = 0x00|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                VISS Subscription length [32 Bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                    VISS Subscription String                   |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.7: VISS Subscription using JSON Serialization

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Message ID (0x1000 0x0005) [32 bit]               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Length (0x0000 00A2) [32 bit]                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (0x0000 0000) [32 bit]                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ = 0x02 | RetCode = 0x00|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    VISS Subscription length [32 Bit] : 0x0000 009A            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       Subscription (154 Bytes) :
  {
       "subscriptionId": "45678",
       "data": {
         "path":"Cabin.Door.Row2.Left.IsLocked",
         "dp": {"value": true, "ts":"2020-04-15T13:37:00Z"}
       },
      "ts":"2020-04-15T13:39:00Z"
  }
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.8: Example VISS Subscription using JSON Serialization

Advantages

1. Easier to implement than designing own data structure as the VISS standard already defines requests and responses in JSON.
2. Since it is just a string, it is scalable as internal data can be nested and changed or adapted during runtime without changing the request or response definition on the SOME/IP serialization side.

Disadvantages

1. Can lead to extra space being used for encoding data like field names, or characters like '{', '}', ':', etc. which might already be known and also need to be encoded as a string of UTF-8 characters.

4.6.2 Using pure SOME/IP Serialization

In this approach, each VISS request and response can be encoded using standard SOME/IP data types. The data point (dp) field can be serialized as a union (also known as a variant) as defined by SOME/IP. We describe a mapping with basic data types and arrays for the same using the mapping as shown in Table 4.1. We do not consider padding here.

VSS Datatype	SOME/IP Union Type ID
uint8	1
int8	2
uint16	3
int16	4
uint32	5
int32	6
uint64	7
int64	8
boolean	9
float	10
double	11
string	12
uint8[]	13
int8[]	14
uint16[]	15
int16[]	16
uint32[]	17
int32[]	18
uint64[]	19
int64[]	20

Table 4.1 continued from previous page

VSS Datatype	SOME/IP Union Type ID
boolean[]	21
float[]	22
double[]	23
string[]	24

Table 4.1: VSS Datatype to SOME/IP Union Type ID mapping

All the strings (both dynamic as well as fixed length) are UTF-8 with 3-byte BOM and termination character (0). The SOME/IP header for requests, responses and notifications remains the same as in the JSON serialized version. The payload format changes and hence the lengths of the messages vary as a result. All timestamps are 24 byte fixed strength UTF-8 strings.

Read

Figure 4.9 shows a possible structure of a read request using the SOME/IP based serialization scheme. The fields for length can be 8, 16 or 32 bits. path_length field is used to indicate the length of the path_string field, then comes the optional filter_string with the filter_length indicating the length of this string (length is 0 if not used). And then the same for optional authorization_token string with authorization_length indicating it’s length (0 if not used).

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0001) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length [32 bit]                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (Client ID / Session ID) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ = 0x00 | RetCode = 0x00|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path_length [8 / 16 / 32 Bits]               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path_string                                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 filter_length [8 / 16 / 32 Bits]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  filter_string                                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            authorization_length [8 / 16 / 32 Bits]            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     authorization_token                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

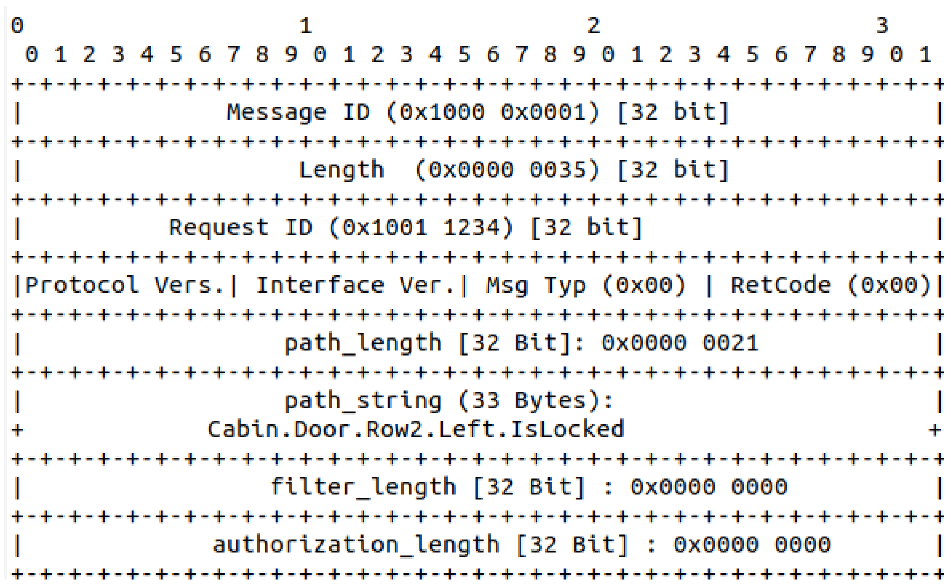
Figure 4.9: VISS Read Request using SOME/IP Serialization

Figure 4.10 shows a possible structure of a read success response serialized using SOME/IP based serialization scheme. The payload contains an array of n data items. Each data item i (where i can be from 1 to n) contains it's own path string denoted by path[i]_string accompanied by a path[i]_length denoting it's length (can be 8, 16 or 32 bits). value[i] is a variant and hence accompanied by value[i]_length and value[i]_type. capture[i]_timestamp_string is the time at which the value[i] was captured. data_length denotes the total length of the data array in bytes starting from path[1]_length, all the way up to capture[n]_timestamp_string. server_timestamp_string is the time at which the request is processed by the server.

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0001) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length [32 bit]                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (Client ID / Session ID) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ (0x80) | RetCode (0x00)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  data_length [8 / 16 / 32 Bits]               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path[1]_length [8 / 16 / 32 Bits]            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path[1]_string                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                value[1]_length [8 / 16 / 32 Bits]             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 value[1]_type [8 / 16 / 32 Bits]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       value[1]                                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 capture[1]_timestamp_string                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                             .
                             .
                             .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path[n]_length [8 / 16 / 32 Bits]            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path[n]_string                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                value[n]_length [8 / 16 / 32 Bits]             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 value[n]_type [8 / 16 / 32 Bits]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      value[n]                                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 capture[n]_timestamp_string                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 server_timestamp_string                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

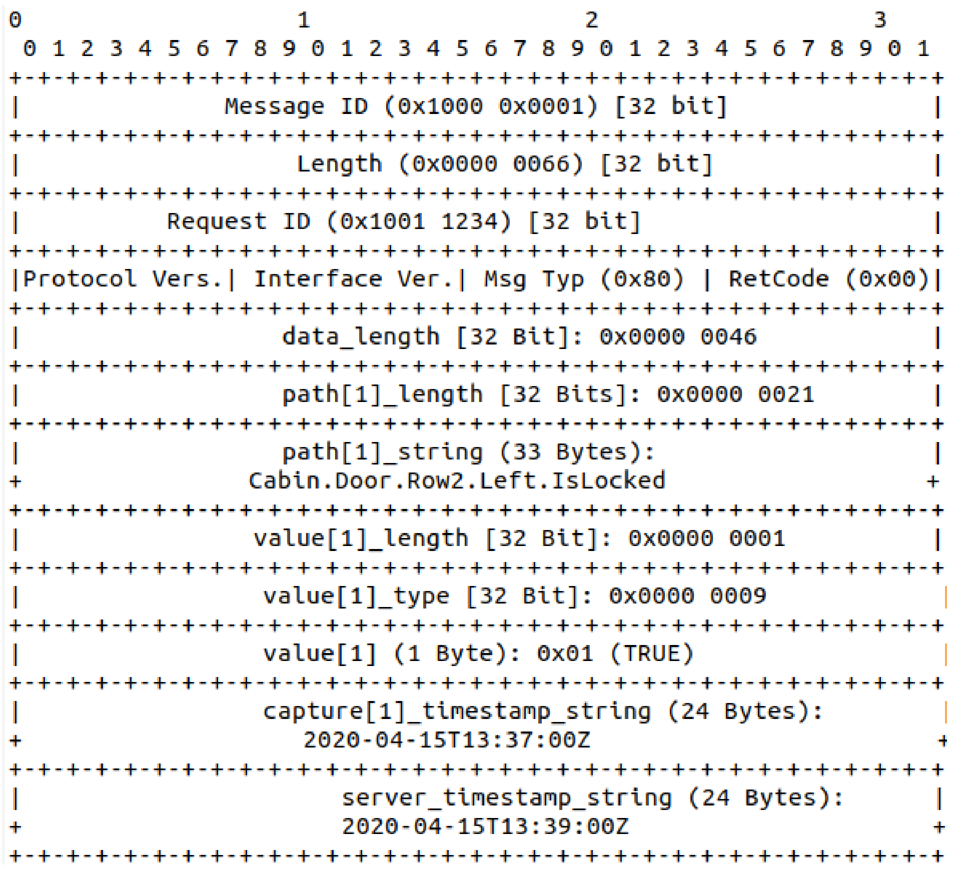
Figure 4.10: VISS Read Success Response using SOME/IP Serialization

Figure 4.11 and Figure 4.12 show examples of a read request and a success response using SOME/IP based serialization scheme. The length fields are chosen to be 32 bits in these examples.



```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0001) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length  (0x0000 0035) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (0x1001 1234) [32 bit]                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ (0x00) | RetCode (0x00)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path_length [32 Bit]: 0x0000 0021            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path_string (33 Bytes):                      |
+            Cabin.Door.Row2.Left.IsLocked                     +
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 filter_length [32 Bit] : 0x0000 0000          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            authorization_length [32 Bit] : 0x0000 0000        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.11: Example VISS Read Request using SOME/IP Serialization



```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0001) [32 bit]               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length (0x0000 0066) [32 bit]                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (0x1001 1234) [32 bit]                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ (0x80) | RetCode (0x00)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  data_length [32 Bit]: 0x0000 0046             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path[1]_length [32 Bits]: 0x0000 0021         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path[1]_string (33 Bytes):                    |
+                Cabin.Door.Row2.Left.IsLocked                   +
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                value[1]_length [32 Bit]: 0x0000 0001           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                value[1]_type [32 Bit]: 0x0000 0009             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                value[1] (1 Byte): 0x01 (TRUE)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                capture[1]_timestamp_string (24 Bytes):         |
+                   2020-04-15T13:37:00Z                         +
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      server_timestamp_string (24 Bytes):       |
+                      2020-04-15T13:39:00Z                      +
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.12: Example VISS Read Success Response using SOME/IP Serialization

Update

Figure 4.13 shows a possible structure of an update request serialized using SOME/IP based serialization scheme. The payload structure is similar to a read request, but with the additional value field where the value to be written is supplied, which is a union / variant and hence accompanied with the value_length and value_type fields.

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0002) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length [32 bit]                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (Client ID / Session ID) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ = 0x00 | RetCode = 0x00|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path_length [8 / 16 / 32 Bits]               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path_string                                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  value_length [8 / 16 / 32 Bits]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 value_type [8 / 16 / 32 Bits]                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           value                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             authorization_length [8 / 16 / 32 Bits]          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     authorization_token                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.13: VISS Update Request using SOME/IP Serialization

Figure 4.14 the possible structure of an update success response serialized using SOME/IP based serialization scheme. The payload consists of the time (server_timestamp_string) at which the request is processed by the server (in this case the time at which the value is updated by the server).

```
 0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |              Message ID (0x1000 0x0002) [32 bit]              |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                   Length [32 bit]                             |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |          Request ID (Client ID / Session ID) [32 bit]         |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |Protocol Vers.| Interface Ver.| Msg Typ (0x80) | RetCode (0x00)|
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                  server_timestamp_string                      |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.14: VISS Update Success Response using SOME/IP Serialization

Figure 4.15 and Figure 4.16 show examples of update request and success response messages. The length fields are chosen to be 32 bits in these examples.

```
 0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |              Message ID (0x1000 0x0002) [32 bit]              |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                   Length  (0x0000 003A) [32 bit]              |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |          Request ID (0x1001 1345) [32 bit]                    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |Protocol Vers.| Interface Ver.| Msg Typ (0x00) | RetCode (0x00)|
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                   path_length [32 Bits]: 0x0000 0021          |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                   path_string (33 Bytes):                     |
 +                 Cabin.Door.Row2.Left.IsLocked                 +
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                 value_length [32 Bit]: 0x0000 0001            |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                  value_type [32 Bit]: 0x0000 0009             |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                  value (1 Byte): 0x00 (FALSE)                 |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |              authorization_length [32 Bit]: 0x0000 0000       |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.15: Example VISS Update Request using SOME/IP Serialization

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0002) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length (0x0000 0020) [32 bit]               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (0x1001 1345) [32 bit]                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ (0x80) | RetCode (0x00)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 server_timestamp_string (24 Bytes):           |
+                     2020-04-15T13:37:00Z                      +
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.16: Example VISS Update Success Response using SOME/IP Serialization

Subscribe

Figure 4.17 shows the possible structure of a subscribe request serialized using SOME/IP based serialization. The payload structure is exactly the same as that of the read request.

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0003) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length [32 bit]                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (Client ID / Session ID) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ = 0x00 | RetCode = 0x00|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path_length [8 / 16 / 32 Bits]               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path_string                                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 filter_length [8 / 16 / 32 Bits]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  filter_string                                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             authorization_length [8 / 16 / 32 Bits]           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     authorization_token                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.17: VISS Subscribe Request using SOME/IP Serialization

Figure 4.18 shows the possible structure of a subscribe success response serialized using SOME/IP based serialization. The payload structure is similar to an update success response with the addition of an additional subscription_id field to provide a unique subscription ID as a 32 bit integer value.

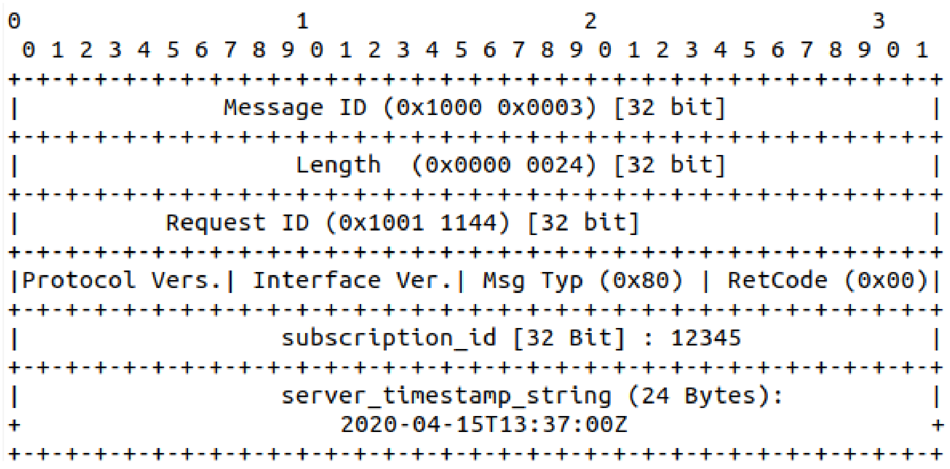
```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0003) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length [32 bit]                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (Client ID / Session ID) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ (0x80) | RetCode (0x00)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  subscription_id                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  server_timestamp_string                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.18: VISS Subscribe Success Response using SOME/IP Serialization

Figure 4.19 and Figure 4.20 show examples of a subscribe request and a success response. The length fields are chosen to be 32 bits in these examples.

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0003) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length  (0x0000 0035) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Request ID (0x1001 1144) [32 bit]                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ (0x00) | RetCode (0x00)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path_length [32 Bit]: 0x0000 0021            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path_string (33 Bytes):                      |
+             Cabin.Door.Row2.Left.IsLocked                     +
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 filter_length [32 Bit] : 0x0000 0000          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             authorization_length [32 Bit] : 0x0000 0000       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.19: Example VISS Subscribe Request using SOME/IP Serialization

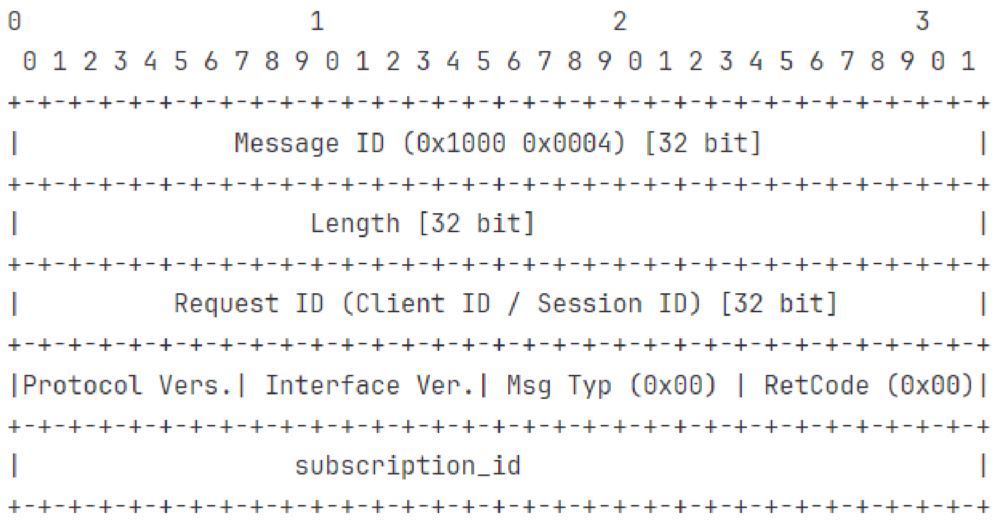


```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0003) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length  (0x0000 0024) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (0x1001 1144) [32 bit]                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ (0x80) | RetCode (0x00)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  subscription_id [32 Bit] : 12345             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  server_timestamp_string (24 Bytes):          |
+                     2020-04-15T13:37:00Z                      +
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.20: Example VISS Subscribe Success Response using SOME/IP Serialization

Unsubscribe

Figure 4.21 shows the possible structure unsubscribe request serialized using SOME/IP based serialization. The payload only consists of a subscription_id where the subscription ID for an active subscription is provided, from which one wishes to unsubscribe.



```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0004) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length [32 bit]                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (Client ID / Session ID) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ (0x00) | RetCode (0x00)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  subscription_id                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.21: VISS Unsubscribe Request using SOME/IP Serialization

Figure 4.22 shows the unsubscribe success response, whose payload structure is exactly the same as that of subscribe success response.

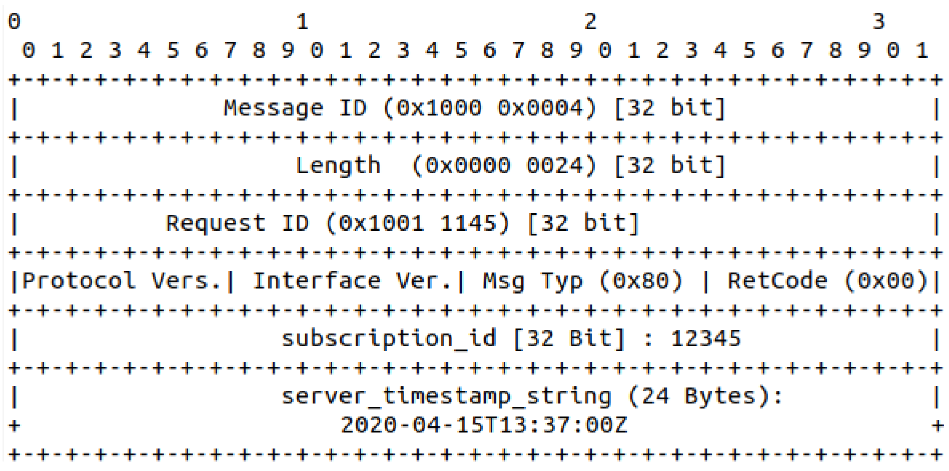
```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0004) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Length [32 bit]                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (Client ID / Session ID) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ (0x80) | RetCode (0x00)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      subscription_id                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    server_timestamp_string                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.22: VISS Unsubscribe Success Response using SOME/IP Serialization

Figure 4.23 and Figure 4.24 show examples of a unsubscribe request and a success response. The length fields are chosen to be 32 bits in these examples.

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0004) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length  (0x0000 000C) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Request ID (0x1001 1145) [32 bit]                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ (0x00) | RetCode (0x00)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  subscription_id [32 Bit] : 12345             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.23: Example VISS Unsubscribe Request using SOME/IP Serialization

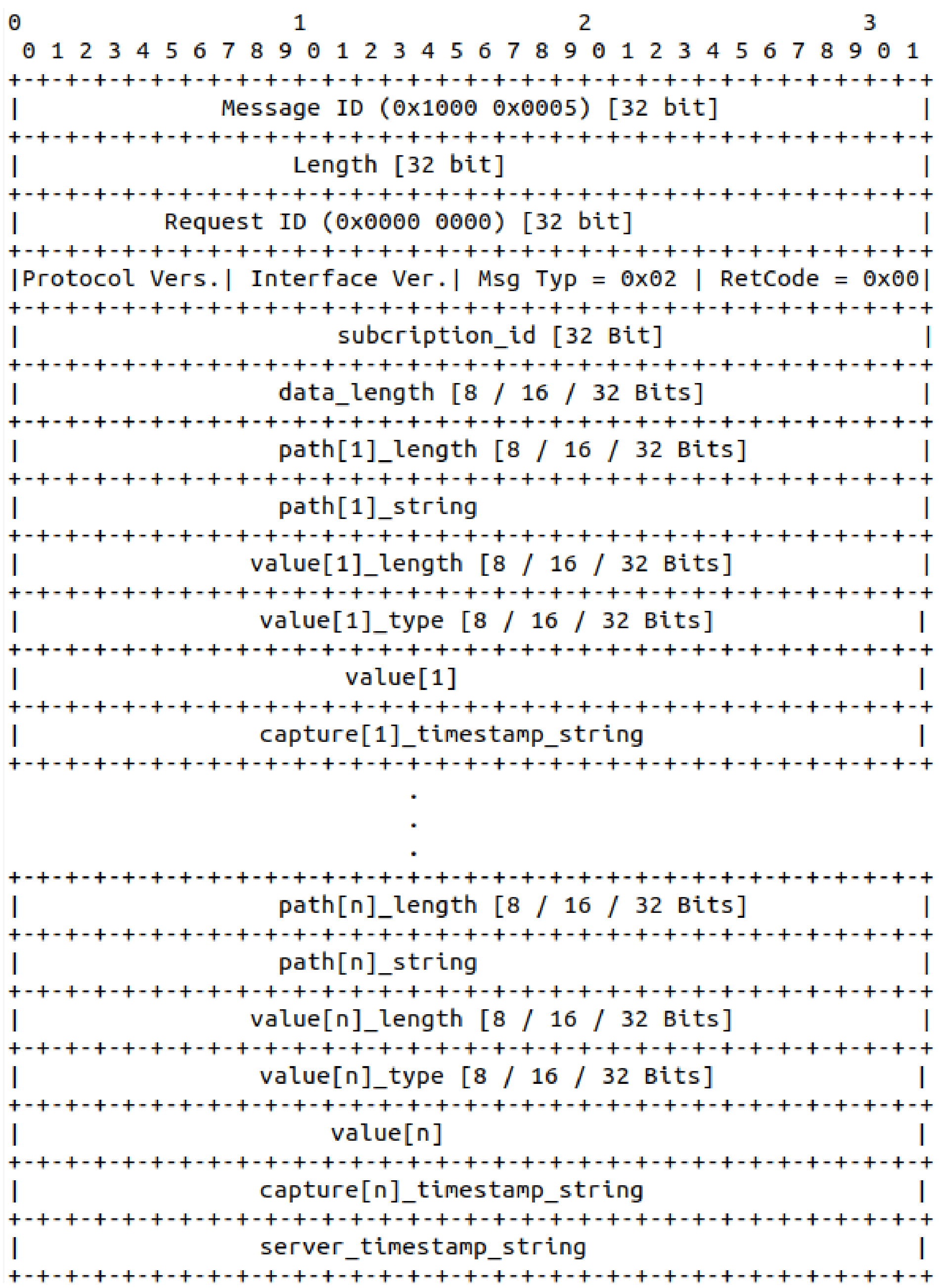


```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0004) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length  (0x0000 0024) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (0x1001 1145) [32 bit]                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ (0x80) | RetCode (0x00)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  subscription_id [32 Bit] : 12345             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  server_timestamp_string (24 Bytes):          |
+                      2020-04-15T13:37:00Z                     +
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.24: Example VISS Unsubscribe Success Response using SOME/IP Serialization

Subscription

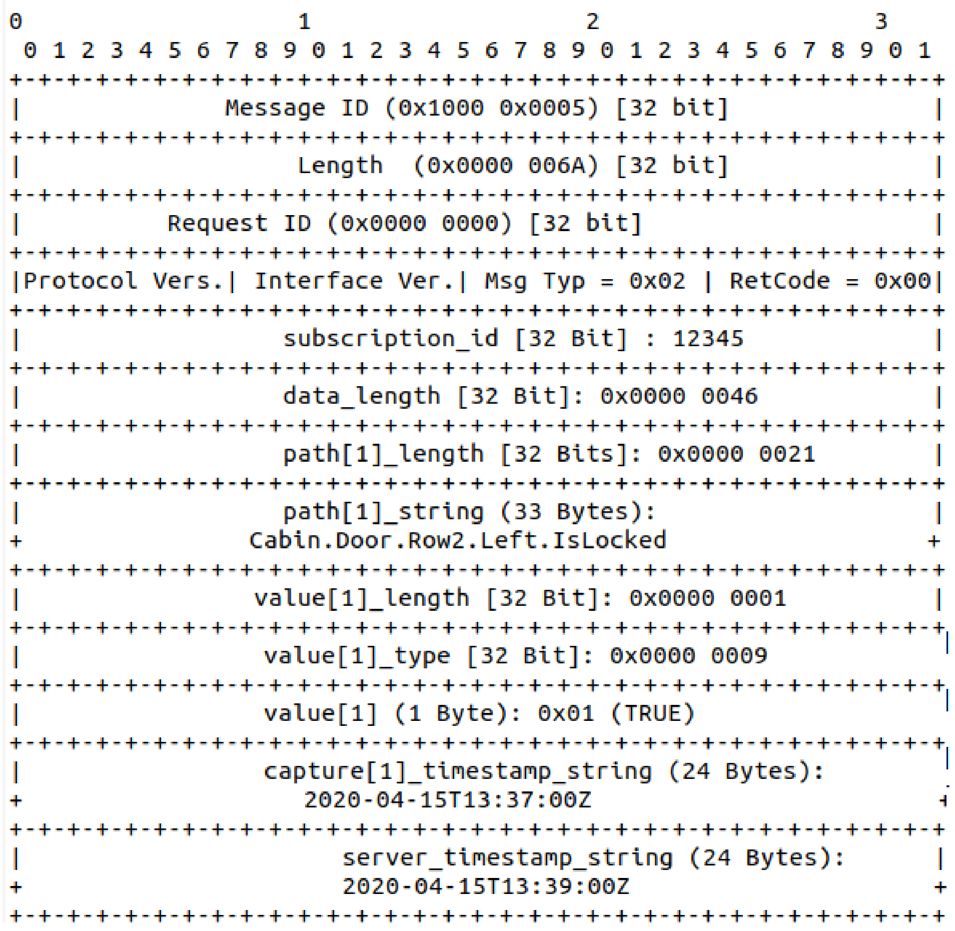
Figure 4.25 shows the possible structure of a subscription notification serialized using SOME/IP based serialization. The payload structure is similar to a read success response with the addition of an additional subscription_id field to provide a subscriptionId as a 32 bit integer value.



```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0005) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length [32 bit]                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (0x0000 0000) [32 bit]                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ = 0x02 | RetCode = 0x00|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      subcription_id [32 Bit]                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  data_length [8 / 16 / 32 Bits]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path[1]_length [8 / 16 / 32 Bits]            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path[1]_string                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 value[1]_length [8 / 16 / 32 Bits]            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 value[1]_type [8 / 16 / 32 Bits]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       value[1]                                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 capture[1]_timestamp_string                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                              .
                              .
                              .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path[n]_length [8 / 16 / 32 Bits]            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path[n]_string                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 value[n]_length [8 / 16 / 32 Bits]            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 value[n]_type [8 / 16 / 32 Bits]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      value[n]                                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 capture[n]_timestamp_string                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 server_timestamp_string                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.25: Example VISS Subscription using SOME/IP Serialization

Figure 4.26 shows an example of a subscription notification. The length fields are chosen to be 32 bits in this example.

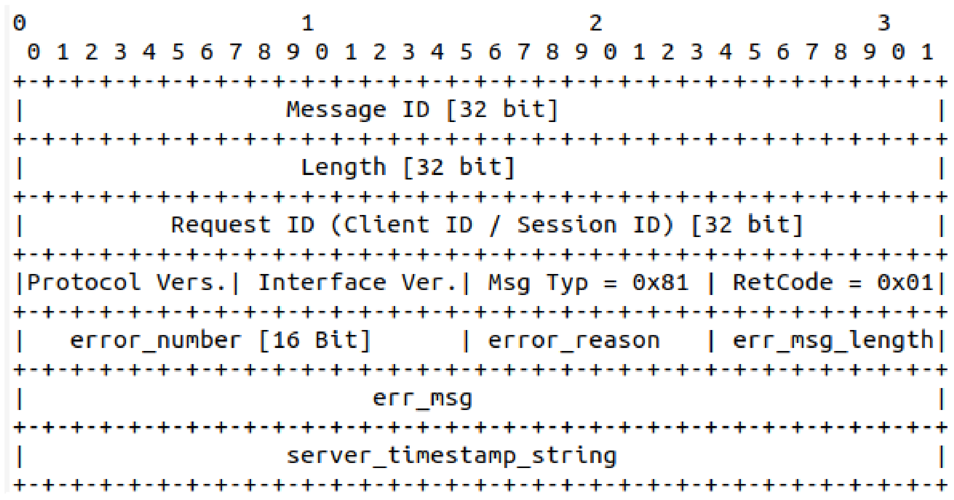


```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0005) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length  (0x0000 006A) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (0x0000 0000) [32 bit]                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ = 0x02 | RetCode = 0x00|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  subscription_id [32 Bit] : 12345             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  data_length [32 Bit]: 0x0000 0046            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path[1]_length [32 Bits]: 0x0000 0021        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  path[1]_string (33 Bytes):                   |
+               Cabin.Door.Row2.Left.IsLocked                   +
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                value[1]_length [32 Bit]: 0x0000 0001          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 value[1]_type [32 Bit]: 0x0000 0009            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 value[1] (1 Byte): 0x01 (TRUE)                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 capture[1]_timestamp_string (24 Bytes):        |
+                    2020-04-15T13:37:00Z                        +
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       server_timestamp_string (24 Bytes):     |
+                       2020-04-15T13:39:00Z                    +
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.26: Example VISS Subscription using SOME/IP Serialization

Error Response

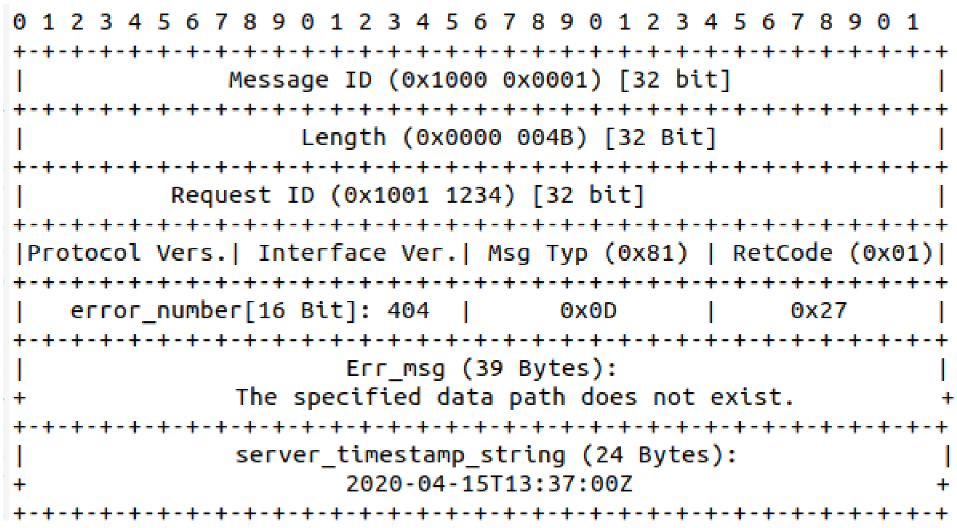
Figure 4.27 shows a possible structure for an error response. The error_number field is chosen to be 16 bits, the error_reason field can map to a error reason stored in an enum and hence only 8 bits assigned here. The error message (err_msg) needs to represent a proper message and hence a dynamic UTF-8 string with 8 bits err_msg_length field indicating it's length. The message is concluded by a server_timestamp_string field.



```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Message ID [32 bit]                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length [32 bit]                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (Client ID / Session ID) [32 bit]         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ = 0x81 | RetCode = 0x01|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   error_number [16 Bit]      | error_reason   | err_msg_length|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        err_msg                                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  server_timestamp_string                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.27: VISS Error Response using SOME/IP Serialization

Figure 4.28 shows an example error response for the VISS read request shown in Figure 4.11. The error reason is chosen to be invalid_path (in this example mapped to an enumeration with value 0x0D for invalid_path) as described in section 4.1 of [13].



```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Message ID (0x1000 0x0001) [32 bit]              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Length (0x0000 004B) [32 Bit]               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Request ID (0x1001 1234) [32 bit]                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Protocol Vers.| Interface Ver.| Msg Typ (0x81) | RetCode (0x01)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   error_number[16 Bit]: 404  |      0x0D      |      0x27     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Err_msg (39 Bytes):                      |
+              The specified data path does not exist.          +
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              server_timestamp_string (24 Bytes):              |
+                     2020-04-15T13:37:00Z                      +
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.28: Example VISS Error Response using SOME/IP Serialization

Advantages

1. Can save space if the structure of data is already known as field names and special characters need not be encoded separately.

Disadvantages

1. Need to design separate request and response structures for all supported VISS operations.

4.6.3 Comparison Between Both Serialization Formats

Table 4.2 compares the payload sizes of the examples given in Section 4.6.1 and Section 4.6.2. As mentioned before the SOME/IP based scheme is more efficient in terms of storage than the JSON based scheme which is evident from the comparison But if any big structural change happens, then it is easier to work with the JSON based serialization, as the fundamental type remains a string.

	JSON	**SOME/IP**
Read Request	48 Bytes	45 Bytes
Read Response	133 Bytes	94 Bytes
Update Request	62 Bytes	50 Bytes
Update Response	40 Bytes	24 Bytes
Subscribe Request	48 Bytes	45 Bytes
Subscribe Response	63 Bytes	28 Bytes
Unsubscribe Request	34 Bytes	4 Bytes
Unsubscribe Response	63 bytes	28 Bytes
Subscription	158 Bytes	98 Bytes
Error Response	136 Bytes	67 bytes

Table 4.2: Comparison Between the Payload Sizes of Messages Based on Serialization Type

4.7 Address Resolution

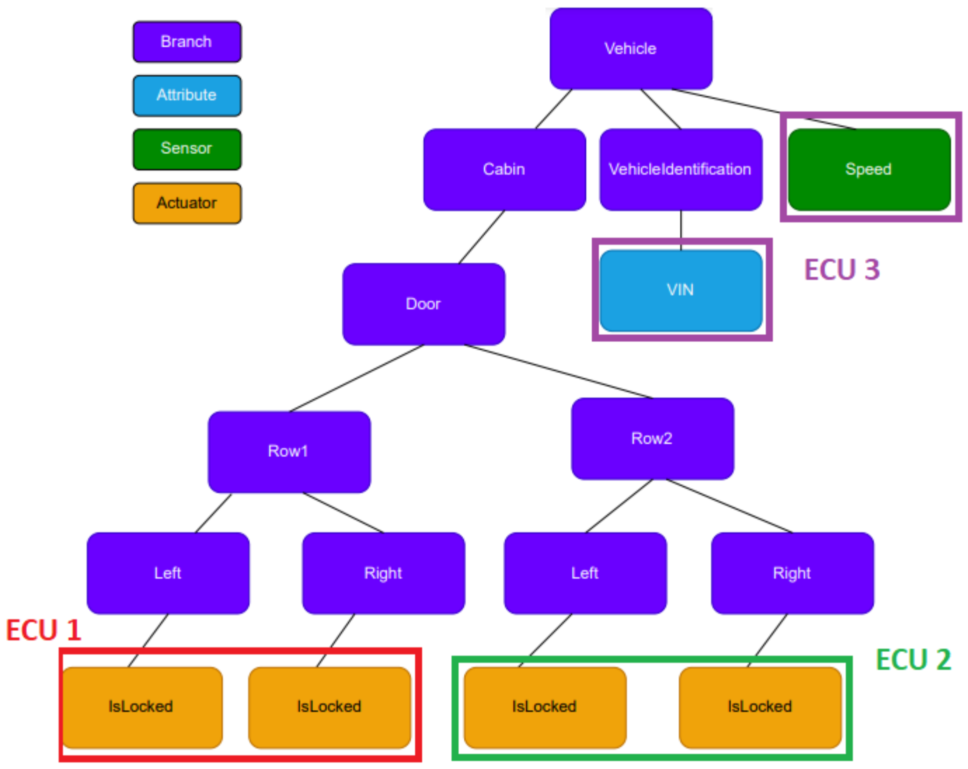


Figure 4.29: Example VSS Tree with Partitioning

In Figure 4.29, we take the advantage of the distributed nature of ECUs present in a vehicle. The signals are distributed over ECU 1, ECU 2 and ECU 3. ECU 1 contains the signals for the first row namely, **Vehicle.Cabin.Door.Row1.Left.IsLocked** and **Vehicle.Cabin.Door.Row1.Right.IsLocked**. Similarly ECU 2 contains the same signals but for the second row namely, **Vehicle.Cabin.Door.Row2.Left.IsLocked** and **Vehicle.Cabin.Door.Row2.Right.IsLocked**. ECU 3 contains **Vehicle.Speed** and **Vehicle.VehicleIdentification.VIN**.

We came up with three approaches for resolving the server responsible for a particular signal. To make the sequence diagrams easier to comprehend, we only consider a subset of the given tree with only 2 ECUs named Server_A and Server_B containing **Vehicle.Cabin.Door.Row1.Left.IsLocked** and **Vehicle.Cabin.Door.Row2.Left.IsLocked** signals respectively. In the sequence diagrams, the **User** can be some general application like in an IVI unit (most likely an user application), which wants to send and receive VISS requests and responses, without getting into the details of implementation of the SOME/IP transport. **Client** is the actual implementation of the SOME/IP client responsible for receiving requests from User and forwarding them to the suitable VISS server (in this case either Server_A or Server_A), as well as receiving responses from the server using SOME/IP as the network binding, and forwarding them to the User.

We have used the approach described in Section 4.2.2 for the below described address resolution approaches. The Service IDs and Method IDs for the VISS operations over SOME/IP are described in the mapping given in Table 4.3.

Service ID	**Method ID**	**VISS Operation**
0x1000	0x0001	Read
0x1000	0x0002	Update
0x1000	0x0003	Subscribe
0x1000	0x0004	Unsubscribe
0x1000	0x0005	Notification
0x2000	0x0001	Read (via Proxy)
0x2000	0x0002	Update (via Proxy)
0x2000	0x0003	Subscribe (via Proxy)
0x2000	0x0004	Unsubscribe (via Proxy)
0x2000	0x0005	Notification (via Proxy)

Table 4.3: SOME/IP Method ID to VISS Operation Mapping

4.7.1 Approach 1: No Service Discovery

In this approach, a client side mapping (probably in the form of a manifest) is available. Each client uses this local mapping to resolve the server responsible for providing access to the required signal.

In Figure 4.30, the User requests to read the signal **Vehicle.Cabin.Door.Row2.Left.IsLocked**. It sends a VISS read request to the client, which in turn, receives this request in the form of a standard VISS request as defined by it's API. The client then looks up the signal in a local Signal to Server mapping (probably in the form of a manifest). Depending on the result, the Client packs the request into a SOME/IP request message and forwards it to a corresponding Server (In this case Server_B). The Server_B then responds with the corresponding VISS response packed into a SOME/IP response message. The client then receives the response, unpacks it and sends it back to the User as a standard VISS read response.

Advantages

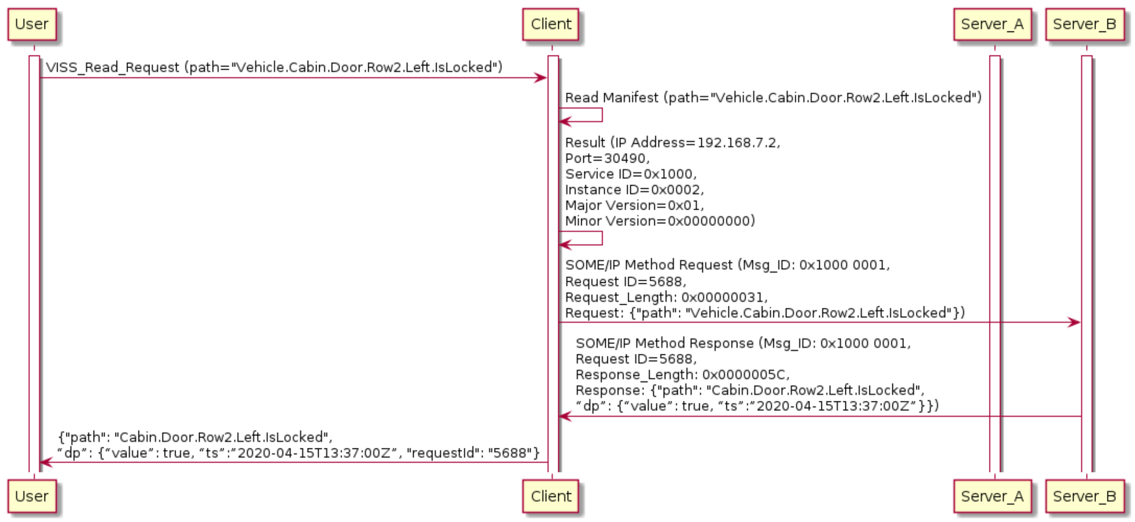
1. No need of separate service discovery phase as the endpoint information is available locally.
2. No upper limit on number of signals offered by a service instance.

Disadvantages

1. Inflexible in case of any change to the network structure. Mappings need to be manually configured again for each client separately.

2. Similarly, in case of changes to the network during runtime, for example via partial networking, where some parts of network are not used, can lead to timeouts on the client side if the server becomes unavailable as the client tries to look for only the preconfigured endpoint.

3. The client knows the endpoint information of the servers directly, which may not be suitable in the security context for certain deployment scenarios.



```
VISS_Read_Request (path="Vehicle.Cabin.Door.Row2.Left.IsLocked")

Read Manifest (path="Vehicle.Cabin.Door.Row2.Left.IsLocked")

Result (IP Address=192.168.7.2,
Port=30490,
Service ID=0x1000,
Instance ID=0x0002,
Major Version=0x01,
Minor Version=0x00000000)

SOME/IP Method Request (Msg_ID: 0x1000 0001,
Request ID=5688,
Request_Length: 0x00000031,
Request: {"path": "Vehicle.Cabin.Door.Row2.Left.IsLocked"})

SOME/IP Method Response (Msg_ID: 0x1000 0001,
Request ID=5688,
Response_Length: 0x0000005C,
Response: {"path": "Cabin.Door.Row2.Left.IsLocked",
"dp": {"value": true, "ts":"2020-04-15T13:37:00Z"}})

{"path": "Cabin.Door.Row2.Left.IsLocked",
"dp": {"value": true, "ts":"2020-04-15T13:37:00Z", "requestId": "5688"}
```

Figure 4.30: Address Resolution Approach 1

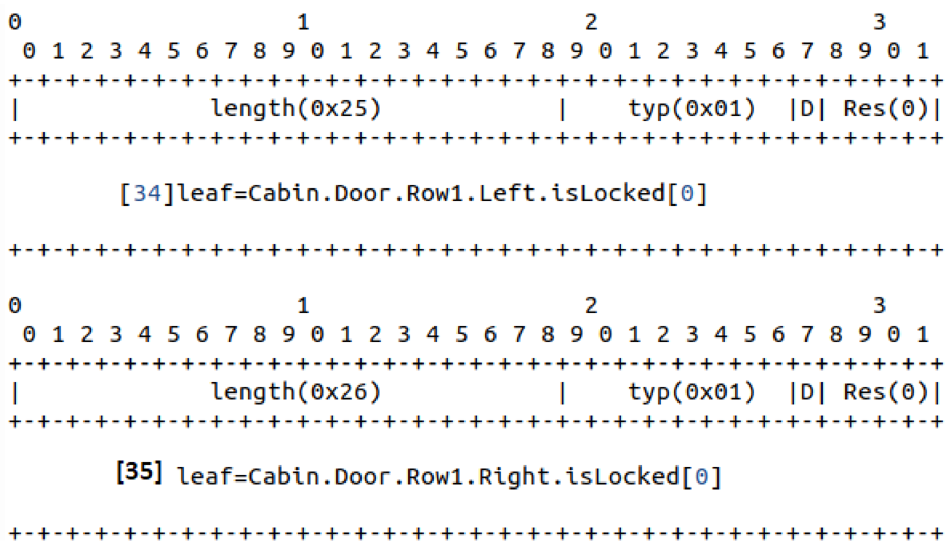
4.7.2 Approach 2: Decentralized Service Discovery and Decentralized Messaging

In this approach, all servers responsible for particular signals, announce these signals during the service discovery phase in the OfferService messages using one configuration option per signal (for example, (Figure 4.31)). The clients can then directly compare each offer with the signal they want and if found in one of the configuration options, can then choose the particular server for accessing the signal. In case of multiple servers offering the same signal, the one with the first match will be used.

In Figure 4.33, instead of the client looking into a local signal to server mapping to find the corresponding server, it looks at the configuration options of servers referenced by their respective OfferService entries during the service discovery phase. When a corresponding configuration option has the signal that the client is looking for, it looks into the endpoint option (in this case IPv4) referenced by the corresponding OfferService entry. This way the client knows where to send SOME/IP requests and receive responses from. The client sees that the given signal is available at Server_B indicated by the configuration string **"leaf=Vehicle.Cabin.Door.Row2.Left.IsLocked"**. After Service Discovery, the rest of the process is the same as previous approach.

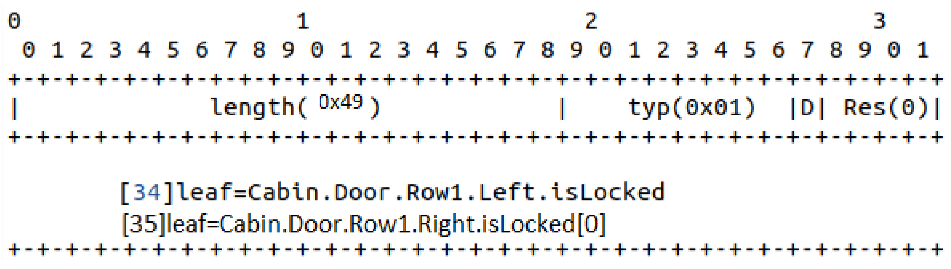
One point to note here is that, SOME/IP-SD allows only a single configuration option to be sent per OfferService message. But if the condition is relaxed then using UDP one should be able to send 65507 bytes of payload with fragmentation (using IP fragmentation [39]) and 1472 bytes without fragmentation. This means that after subtracting the SOME/IP header size (16 bytes), SOME/IP-SD message with length arrays for both entries and options, OfferService entry and an

IPv4 endpoint option (a total of 40 bytes), we are left with 65451 bytes for configuration options with fragmentation and 1416 bytes without fragmentation. So if assuming configuration options of length each 40 bytes are used, then one should be able to transmit a total of 1636 and 35 signals per service instance for both cases respectively. [9] states SOME/IP Transport Protocol (SOME/IP-TP) which allows to send bigger UDP messages limiting the size of a segment to 1400 bytes, meaning 1392 bytes usable by SOME/IP message. But since, SOME/IP-SD standard allows duplicate keys in different entries, one can also put the signals as separate entries within the same configuration option like [length1]leaf=signal1[length2]leaf=signal2....[0] (Figure 4.32) and still use the UDP limits and follow the SOME/IP-SD standard.



```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             length(0x25)            |    typ(0x01)  |D| Res(0)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

        [34]leaf=Cabin.Door.Row1.Left.isLocked[0]

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             length(0x26)            |    typ(0x01)  |D| Res(0)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

        [35] leaf=Cabin.Door.Row1.Right.isLocked[0]

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.31: Separate Configuration Option Per Signal



```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             length( 0x49 )          |    typ(0x01)  |D| Res(0)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

        [34]leaf=Cabin.Door.Row1.Left.isLocked
        [35]leaf=Cabin.Door.Row1.Right.isLocked[0]
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.32: Single Configuration Option for all Signals

Advantages

1. More flexible as compared to the 1st approach due to the fact that each server now manages it's own list of signals and hence more dynamic as the list can now be directly updated on the offering server itself.
2. More tolerant to failures as compared to the previous approach due to it's distributed working.

Disadvantages

1. OfferService messages can contain a lot of data depending on the number of signals and can affect the bandwidth in case of cyclic messages if the bandwidth is scarce.
2. UDP has upper limits on message sizes.
3. The client knows the endpoint information of the servers directly, which may not be suitable in the security context for certain deployment scenarios.

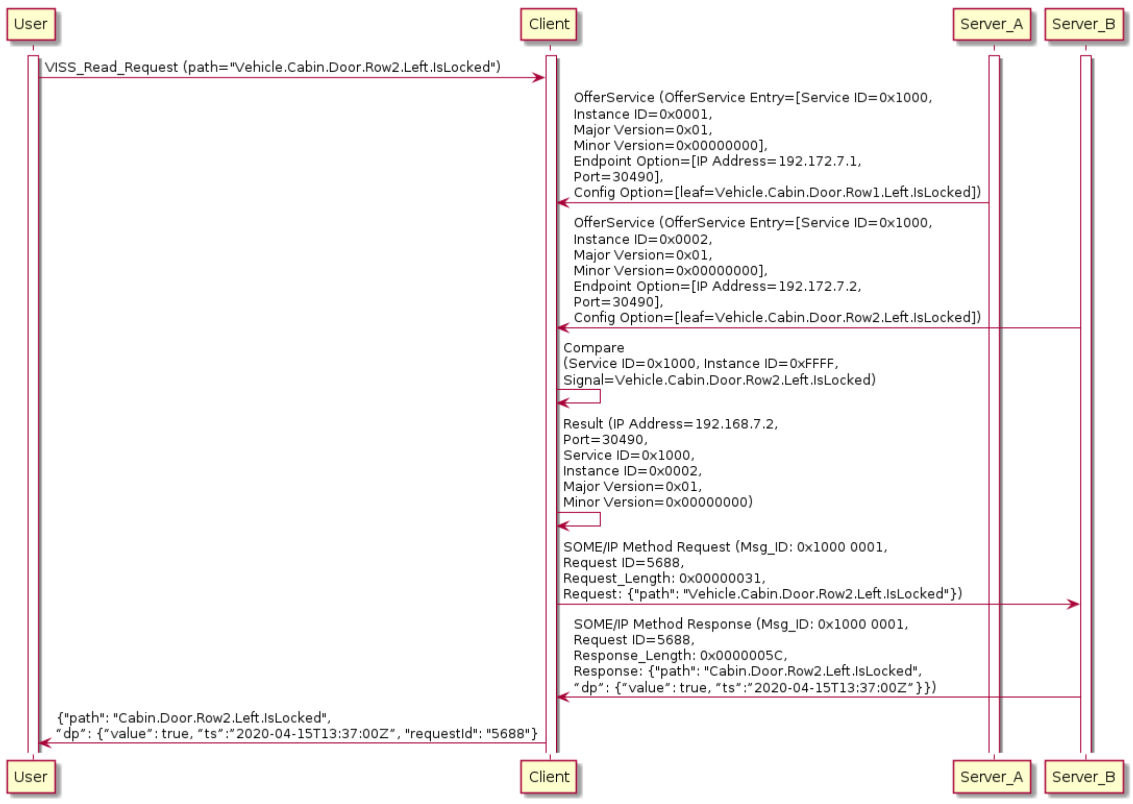

User
Client
Server_A
Server_B
VISS_Read_Request (path="Vehicle.Cabin.Door.Row2.Left.IsLocked")
OfferService (OfferService Entry=[Service ID=0x1000, Instance ID=0x0001, Major Version=0x01, Minor Version=0x00000000], Endpoint Option=[IP Address=192.172.7.1, Port=30490], Config Option=[leaf=Vehicle.Cabin.Door.Row1.Left.IsLocked])
OfferService (OfferService Entry=[Service ID=0x1000, Instance ID=0x0002, Major Version=0x01, Minor Version=0x00000000], Endpoint Option=[IP Address=192.172.7.2, Port=30490], Config Option=[leaf=Vehicle.Cabin.Door.Row2.Left.IsLocked])
Compare (Service ID=0x1000, Instance ID=0xFFFF, Signal=Vehicle.Cabin.Door.Row2.Left.IsLocked)
Result (IP Address=192.168.7.2, Port=30490, Service ID=0x1000, Instance ID=0x0002, Major Version=0x01, Minor Version=0x00000000)
SOME/IP Method Request (Msg_ID: 0x1000 0001, Request ID=5688, Request_Length: 0x00000031, Request: {"path": "Vehicle.Cabin.Door.Row2.Left.IsLocked"})
SOME/IP Method Response (Msg_ID: 0x1000 0001, Request ID=5688, Response_Length: 0x0000005C, Response: {"path": "Cabin.Door.Row2.Left.IsLocked", "dp": {"value": true, "ts":"2020-04-15T13:37:00Z"}})
{"path": "Cabin.Door.Row2.Left.IsLocked", "dp": {"value": true, "ts":"2020-04-15T13:37:00Z", "requestId": "5688"}
User
Client
Server_A
Server_B

Figure 4.33: Address Resolution Approach 2

4.7.3 Approach 3: Centralized Service Discovery and Centralized Messaging

This approach is a hybrid of approach 1 and approach 2. The Service Discovery mechanism remains the same as approach 2, but instead of client directly discovering the servers via SOME/IP-SD messages, this task is given to a separate Proxy which does this Service Discovery and maintains an internal Signal to Server mapping, so that the client only sees the Proxy as the VISS server. The Proxy uses a separate Service ID, but the rest of the service design including method signatures remain the same as that of the VISS servers. The Proxy's sole purpose is to take the SOME/IP requests from the clients and after looking up the Signal to Server mapping, change the request header to the one required by the server and do the reverse for responses received from server and routing them back to client.

In Figure 4.34, the User sends the VISS read request to read the signal **Vehicle.Cabin.Door.Row2.Left.**
IsLocked the same way as the previous two approaches. The client does Service Discovery only for the Proxy and once discovered packs the VISS request into a SOME/IP request message with service ID of the Proxy (in this case 0x2000). The method ID and other parameters remain the same as the previous approaches. The Proxy is a long running process (probably a daemon), which monitors for OfferService entries with configuration and endpoint options like the client in approach 2 did and maintains a local Signal to Server mapping which it keeps updating on the basis of received SOME/IP-SD messages from servers. It also has to maintain a mapping to resolve the responses received from servers back to the respective clients. Upon receiving the SOME/IP request from client, the Proxy does a lookup for the respective server (in this case Server_B), changes the Service ID to the one required by the server (In this case 0x1000) and sends the request to the server. Upon receiving response from the server, the Proxy looks up which client it is for, changes the Service ID of the response again to 0x2000, and forwards it to the client. The client then unpacks the SOME/IP response message and sends the standard VISS response to the User.

Advantages

1. More secure than approach 1 and approach 2, as the access to internal VISS servers is managed at a central place (Proxy) and the clients do not know the endpoint information of these servers directly. In fact, the servers can be placed in a different multicast group as long as the proxy can listen to their SOME/IP-SD messages containing the OfferService entries, configuration options and endpoint options.
2. The clients do not have to process configuration options of multiple servers by themselves as it is done by the proxy itself.

Disadvantages

1. OfferService messages can contain a lot of data depending on the number of signals and can affect the bandwidth in case of cyclic messages if the bandwidth is scarce.
2. UDP has upper limits on message sizes.
3. The proxy is the central point of failure.

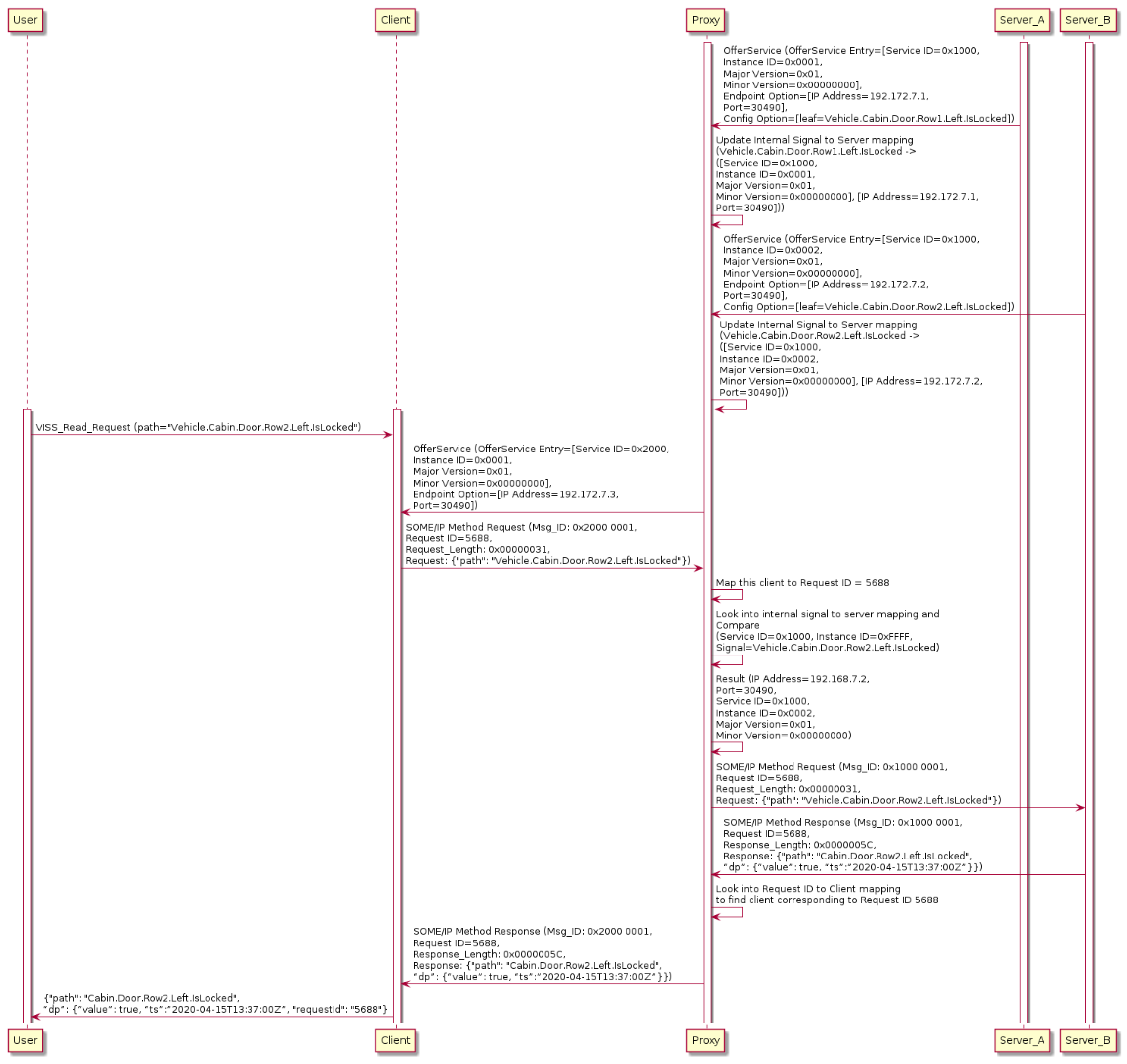

User
Client
Proxy
Server_A
Server_B
OfferService (OfferService Entry=[Service ID=0x1000,
Instance ID=0x0001,
Major Version=0x01,
Minor Version=0x00000000],
Endpoint Option=[IP Address=192.172.7.1,
Port=30490],
Config Option=[leaf=Vehicle.Cabin.Door.Row1.Left.IsLocked])
Update Internal Signal to Server mapping
(Vehicle.Cabin.Door.Row1.Left.IsLocked ->
([Service ID=0x1000,
Instance ID=0x0001,
Major Version=0x01,
Minor Version=0x00000000], [IP Address=192.172.7.1,
Port=30490]))
OfferService (OfferService Entry=[Service ID=0x1000,
Instance ID=0x0002,
Major Version=0x01,
Minor Version=0x00000000],
Endpoint Option=[IP Address=192.172.7.2,
Port=30490],
Config Option=[leaf=Vehicle.Cabin.Door.Row2.Left.IsLocked])
Update Internal Signal to Server mapping
(Vehicle.Cabin.Door.Row2.Left.IsLocked ->
([Service ID=0x1000,
Instance ID=0x0002,
Major Version=0x01,
Minor Version=0x00000000], [IP Address=192.172.7.2,
Port=30490]))
VISS_Read_Request (path="Vehicle.Cabin.Door.Row2.Left.IsLocked")
OfferService (OfferService Entry=[Service ID=0x2000,
Instance ID=0x0001,
Major Version=0x01,
Minor Version=0x00000000],
Endpoint Option=[IP Address=192.172.7.3,
Port=30490])
SOME/IP Method Request (Msg_ID: 0x2000 0001,
Request ID=5688,
Request_Length: 0x00000031,
Request: {"path": "Vehicle.Cabin.Door.Row2.Left.IsLocked"})
Map this client to Request ID = 5688
Look into internal signal to server mapping and
Compare
(Service ID=0x1000, Instance ID=0xFFFF,
Signal=Vehicle.Cabin.Door.Row2.Left.IsLocked)
Result (IP Address=192.168.7.2,
Port=30490,
Service ID=0x1000,
Instance ID=0x0002,
Major Version=0x01,
Minor Version=0x00000000)
SOME/IP Method Request (Msg_ID: 0x1000 0001,
Request ID=5688,
Request_Length: 0x00000031,
Request: {"path": "Vehicle.Cabin.Door.Row2.Left.IsLocked"})
SOME/IP Method Response (Msg_ID: 0x1000 0001,
Request ID=5688,
Response_Length: 0x0000005C,
Response: {"path": "Cabin.Door.Row2.Left.IsLocked",
“dp”: {“value”: true, “ts”:”2020-04-15T13:37:00Z”}})
Look into Request ID to Client mapping
to find client corresponding to Request ID 5688
SOME/IP Method Response (Msg_ID: 0x2000 0001,
Request ID=5688,
Response_Length: 0x0000005C,
Response: {"path": "Cabin.Door.Row2.Left.IsLocked",
“dp”: {“value”: true, “ts”:”2020-04-15T13:37:00Z”}})
{"path": "Cabin.Door.Row2.Left.IsLocked",
“dp”: {“value”: true, “ts”:”2020-04-15T13:37:00Z”, "requestId": "5688"}
User
Client
Proxy
Server_A
Server_B

Figure 4.34: Address Resolution Approach 3

5 Realization

For the implementation we have only considered a SOME/IP client and server to make it easy to implement and evaluate our design. The basic idea is that the client observes all the offers with the configuration options. Depending on the signal to be read, updated, or subscribed to, the client compares the signal described in configuration option and sends the request to the corresponding server. One can also instantiate the client in such a way that it looks into a static signal to server mapping to discover servers instead of comparing configuration options.

The implementation was done on C++ using Adaptive MICROSAR [69] libraries provided by Vector Informatik GmbH and the JSON library from Microsoft's C++ Rest SDK [57].

5.1 Components

Figure 5.3 shows the component diagram for the server **(SOMEIPVISSServer)** and Figure 5.4 for the client **(SOMEIPVISSClient)** respectively. The common components are as follows:

1. **ServiceDiscoveryEndpoint:** This is responsible for maintaining an Unicast UDP endpoint and joining a multicast group for sending and receiving SOME/IP-SD messages. In our case for sending and receiving OfferService messages on the server and the client respectively.

2. **SDMessageProcessorInterface:** A reference to an object implementing this interface is passed to the constructor of ServiceDscoveryEndpoint object. The implementing class has to implement a **process** function using which the implementing class can apply its logic to a received SOME/IP-SD message. Figure 5.1 shows the classes which implement this interface.

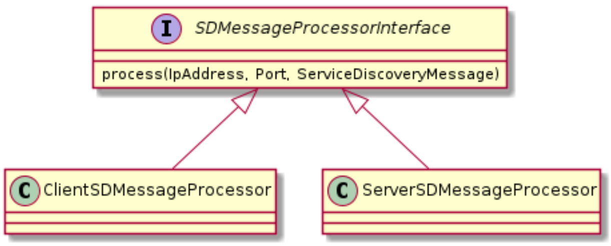

I
SDMessageProcessorInterface
process(IpAddress, Port, ServiceDiscoveryMessage)
C
ClientSDMessageProcessor
C
ServerSDMessageProcessor

Figure 5.1: SDMessageProcessorInterface

3. **TcpEndpoint:** This is responsible for maintaining a local Transmission Control Protocol (TCP) endpoint on the client and server. In case of a server, it implements the TCP server functionality as well. The provided service instance needs to be registered in this object.

4. **TcpConnection:** This is an object which takes the remote and local endpoint and represents a logical TCP connection between the two. This is responsible for forwarding SOME/IP messages to remote client or server. It Also forwards the received requests, responses, or notifications to be processed by client or server. For a client, this is instantiated directly as we know the remote endpoint while connecting (active TCP connection), but for a server this is created by the TcpEndpoint, when it receives the connection request from the client (passive TCP connection). The required service instance needs to be registered in this object.

5. **TcpMessageProcessorInterface:** A reference to an object whose class implements this interface is passed to the TcpConnection and TcpEndpoint constructors. The classes implementing this interface have to implement a **process** function, which is responsible for processing the received message via TCP connection on client or server. Figure 5.2 shows the classes which implement this interface. Note: TcpConnectionSharedPtr is used as an alias to mention a shared pointer to a TcpConnection.

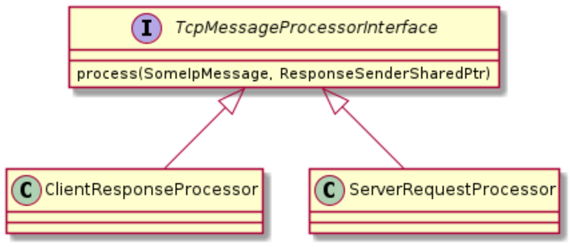

I TcpMessageProcessorInterface
process(SomeIpMessage, ResponseSenderSharedPtr)
C ClientResponseProcessor
C ServerRequestProcessor

Figure 5.2: TcpMessageProcessorInterface

6. **Reactor:** This class starts a reactor [65] thread on the client or a server which is then passed to ServiceDiscoveryEndpoint, TcpEndpoint and TcpConnection objects to register their callbacks for the corresponding events in this thread.

7. **TimerManager:** Used for managing timers and repetitive tasks like sending OfferServce messages in fixed intervals and check expiry of tasks.

8. **ReactorThread:** A wrapper taking Reactor and TimerManager objects in it's constructor. It is Used for starting and stopping Reactor in client or server.

9. **VISSError:** A struct which represents a single VISS error.

10. **VISSErrorResponse:** Represents an object encapsulating the VISSError struct.

11. **ErrorResponseJSONSerializer:** Object used to serialize VISSErrorResponse object into JSON.

5.1.1 SOMEIPVISSServer

1. **ServerConfig:** This object is used to pass various configuration parameters to the server instance. These include:

    - **unicast_addr:** The unicast IPv4 address of the server endpoint used for sending and receiving SOME/IP-SD messages via UDP.

- **multicast_addr:** The IPv4 address of the multicast group for SOME/IP-SD.
- **unicast_port:** The port used for the unicast UDP endpoint.
- **multicast_port:** The port used for the multicast.
- **service_id:** Service ID of the provided service instance.
- **instance_id:** Instance ID of the provided service instance.
- **major_version:** Major version of the provided service instance.
- **minor_version:** Minor version of the provided service instance.
- **ttl:** TTL for the OfferService message
- **repetition_time:** Interval between each subsequent OfferService message to be sent.
- **signals:** List of signals (stored as ara::core::String) that are supported by the provided service instance.

2. **ServerSDMessageProcessor:** This class implements the SDMessageProcessorInterface by implementing its process function which is currently blank as we ignored SOME/IP-SD messages from client to server for now.
3. **ServerRequestProcessor:** This class implements the TcpMessageProcessorInterface and processes client method requests and sends back the corresponding responses back to the clients.
4. **ServerSubscriptionMap:** This class is used to keep track of subscriptions based on the signals. It is implemented as a map with keys as signals (represented as ara::core::String) and the value being a list of pairs of SubscriptionId (unsigned 32-bit integer in this implementation) and ResponseSender shared pointers (can be TCP or UDP as long as they implement this interface, but TCP for this implementation). The ResponseSender is made up of TcpEndpoint and TcpConnection in this case and acts as a handle of connections on the server side.

5.1.2 SOMEIPVISSClient

1. **ClientConfig:** This object is used to pass various configuration parameters to the client instance. These include:
   - **unicast_addr:** The unicast IPv4 address of the client endpoint used for sending and receiving SOME/IP-SD messages via UDP.
   - **multicast_addr:** The IPv4 address of the multicast group for SOME/IP-SD.
   - **unicast_port:** The port used for the unicast UDP endpoint.
   - **multicast_port:** The port used for the multicast.
   - **service_id:** The required service instance's service ID.
   - **instance_id:** The required service instance's instance ID.
   - **major_version:** The required service instance's major version.

- **minor_version:** The required service instance's minor version.
- **client_id:** Fixed client ID to be used as a part of request ID when sending requests.
- **signal_to_server_mapping:** A map containing keys as signals (as ara::core::String) and value as a pair of Endpoint object (which contains the IPv4 address and port of the TCP server responsible for handling the respective signal) and the ProvidedServiceInfo object (which contains Service ID, Instance ID, Major Version and Minor Version of the provided service instance).
- **discovery_attempts:** Amount of seconds for which a busy waiting is made for discovering a server in case of dynamic service discovery via configuration options.
- **connection_attempts:** Amount of seconds for which a busy waiting is made for connecting to the remote server.
- **response_attempts:** Amount of seconds for which a busy waiting is made for receiving response from the connected server.

2. **ClientSDMessageProcessor:** This class implements the SDMessageProcessorInterface by implementing its process function which compares the configuration option and the required service parameters given in the constructor to the ones in the OfferService messages, once a match is found whose required service parameters and any one of the configuration option matches the one given during its construction, it saves the remote endpoint information including IPv4 address and port as well as its service instance ID.
3. **ClientResponseProcessor:** This class implements the TcpMessageProcessorInterface and is responsible for processing method responses and notifications coming from the server.
4. **ClientSubscriptionMap:** This object is used to keep track of subscriptions on the client side and is implemented as a map with keys being SubscriptionId (unsigned 32-bit integer) and value being a SubscriptionInfo object (which contains the remote endpoint object, TcpConnection shared pointer, instance ID of the provided service and the user's desired callback function, which is to be called when a subscription is received.
5. **SDConfigurationOption:** This is a SOME/IP-SD configuration option containing the configuration string with key being 'leaf' and the value being the signal name. It is constructed for each separate call of read, update or subscribe methods.

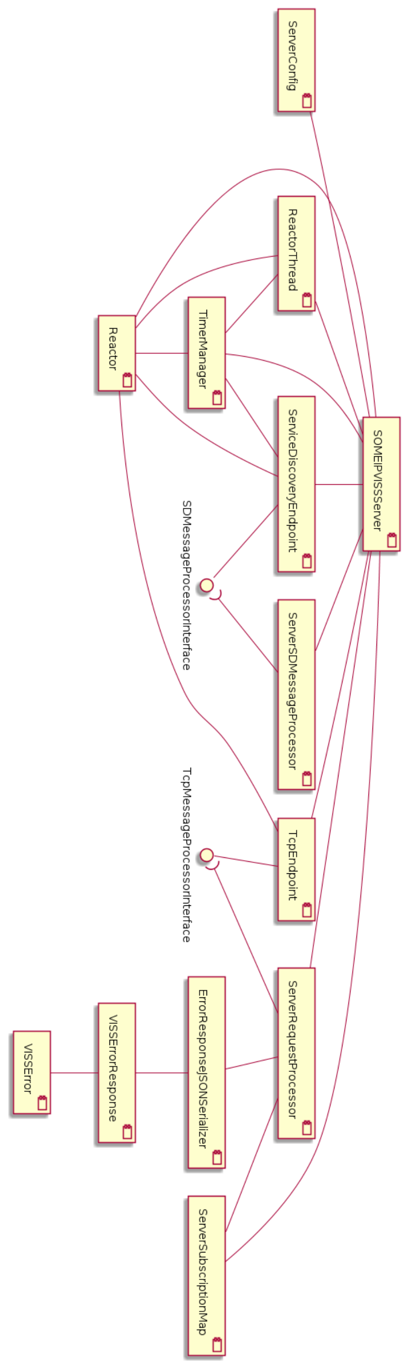


Figure 5.3: SOMEIPVISSServer Component Diagram

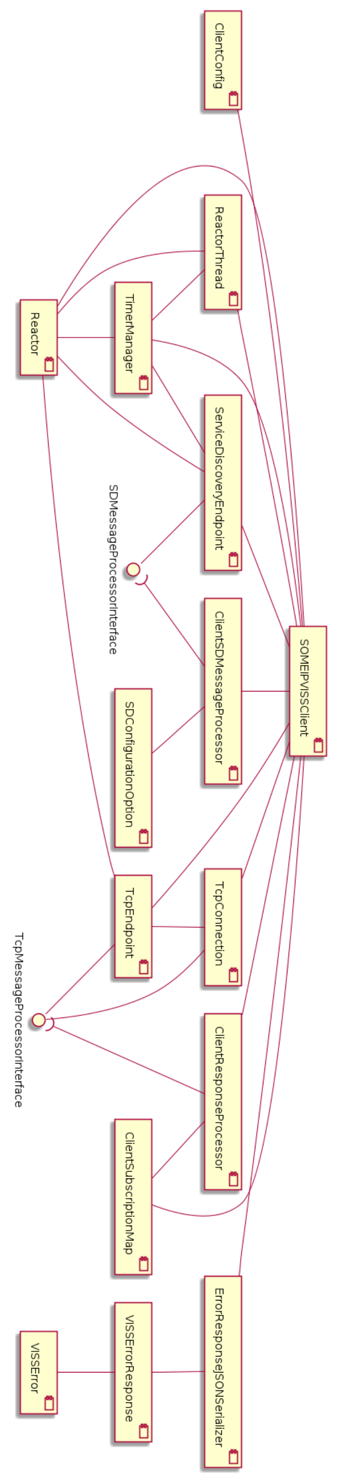


Figure 5.4: SOMEIPVISSClient Component Diagram

5.2 Communication Flows

This section gives insight into some of the important flows required for communication between the SOMEIPVISSClient and the SOMEIPVISSServer instances via SOME/IP with the use of sequence diagrams and pseudo codes (wherever necessary).

5.2.1 Reception of SOME/IP-SD Messages on Client

Figure 5.5 shows how a SOME/IP-SD message is received and processed on the client. The processing happens inside the process() function of ClientSDProcessor object. The function does the following:

1. For each OfferService Entry:
   a) Check if service ID matches the provided one.
   b) Check if the instance ID matches the provided one or the provided one is 0xffff.
   c) Check if the major version matches the provided one or the provided one is 0xff.
   d) Check if the minor version matches the provided one or the provided one is 0xffffffff.
   e) If all the previous conditions fulfilled, then:
      i. Check if the received configuration option (if exists) is equal to the one which is provided, if yes, then:
         A. Store the server endpoint information (IP address, port number) and the instance id.

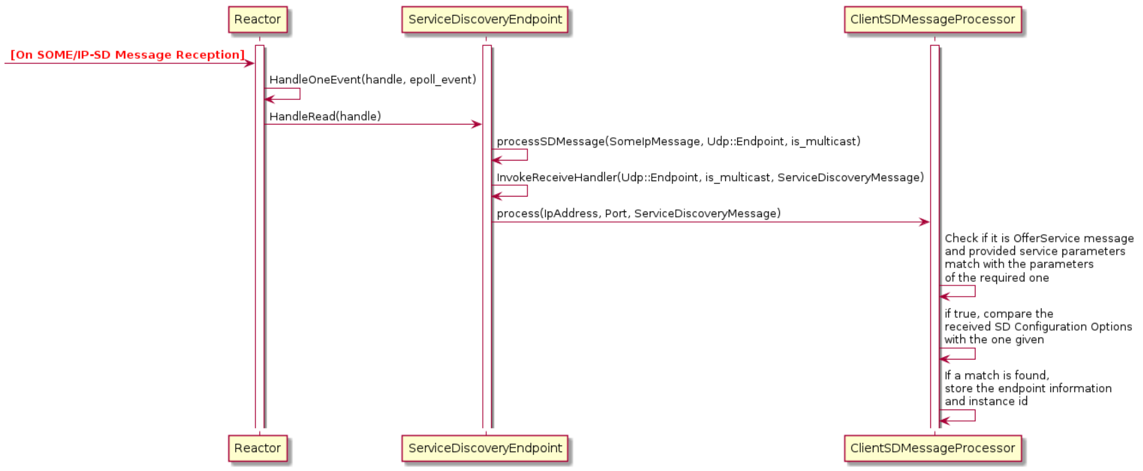

Reactor
ServiceDiscoveryEndpoint
ClientSDMessageProcessor
[On SOME/IP-SD Message Reception]
HandleOneEvent(handle, epoll_event)
HandleRead(handle)
processSDMessage(SomeIpMessage, Udp::Endpoint, is_multicast)
InvokeReceiveHandler(Udp::Endpoint, is_multicast, ServiceDiscoveryMessage)
process(IpAddress, Port, ServiceDiscoveryMessage)
Check if it is OfferService message and provided service parameters match with the parameters of the required one
if true, compare the received SD Configuration Options with the one given
If a match is found, store the endpoint information and instance id
Reactor
ServiceDiscoveryEndpoint
ClientSDMessageProcessor

Figure 5.5: SOME/IP-SD Message Reception on Client

5.2.2 Reception of Response / Notification on Client

Figure 5.6 shows how a client receives and processes a received response or a notification message from the server. The TcpConnection's HandleRead() and ProcessMessage() functions validate the received response if it is coming from the expected service instance and if the format is right before passing it to ClientResponseProcessor. The process() function inside the ClientResponseProcessor object is responsible for checking the message type and based on that dispatch the message to the respective handling sub function. It handles it in the following way:

1. If the message type is 0x80 (Response), then do the following:
    a) If the method ID is 0x01, call the processVISSReadResponse(SomeIpMessage, ResponseSenderSharedPtr) method.
    b) Else if the method ID is 0x02, call the processVISSUpdateResponse(SomeIpMessage, ResponseSenderSharedPtr) method.
    c) Else if the method ID is 0x03, call the processVISSSubscribeResponse(SomeIpMessage, ResponseSenderSharedPtr) method.
    d) Else if the method ID is 0x04, call the processVISSUnsubscribeResponse(SomeIpMessage, ResponseSenderSharedPtr) method.
    e) Else call the processVISSErrorResponse(SomeIpMessage, ResponseSenderSharedPtr) method.
2. Else if the message type is 0x02 (Notification) and method ID is 0x05, then call the processVISSNotification(SomeIpMessage) method.
3. Else, call processVISSError(SomeIpMessage, ResponseSenderSharedPtr) method.

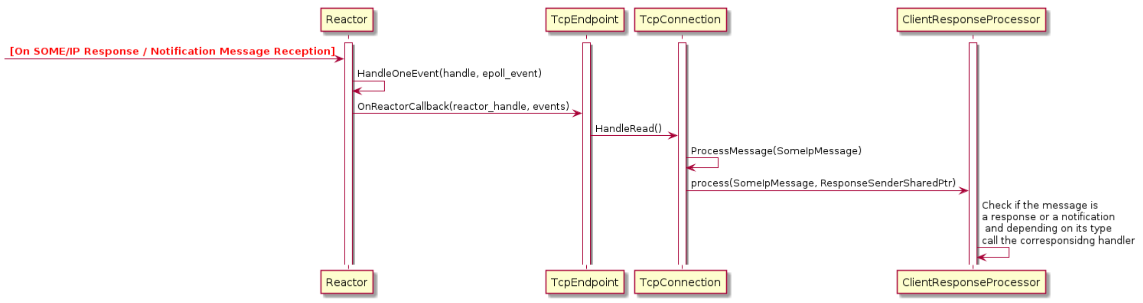

Reactor
TcpEndpoint
TcpConnection
ClientResponseProcessor
[On SOME/IP Response / Notification Message Reception]
HandleOneEvent(handle, epoll_event)
OnReactorCallback(reactor_handle, events)
HandleRead()
ProcessMessage(SomeIpMessage)
process(SomeIpMessage, ResponseSenderSharedPtr)
Check if the message is
a response or a notification
and depending on its type
call the corresponsidng handler
Reactor
TcpEndpoint
TcpConnection
ClientResponseProcessor

Figure 5.6: SOME/IP Response / Notification Reception on Client

5.2.3 Sending Request from Client

Figure 5.7 shows how a request is sent from a client to a server. The client prepares the SOME/IP method request message using the sendRequestPacket() method and then uses the TcpConnection object's Forward() method to send the packet to the server. The sendRequestPacket() method does the following:

1. Set the service ID to ClientConfig.service_id.

2. Set the method ID to method_id.
3. Set the client ID to ClientConfig.client_id.
4. Set the protocol version to 0x1.
5. Set the interface version to ClientConfig.major_version.
6. Set the session ID to a randomly generated ID.
7. Set the length field to 8 + length of the payload (which includes length field of request_string and request_string itself) (in bytes).
8. Set the message type to 0x00 (Request).
9. Set the return code to 0x00 (OK).
10. Serialize the message and send it to server via TcpConnectionSharedPtr's Forward() method.

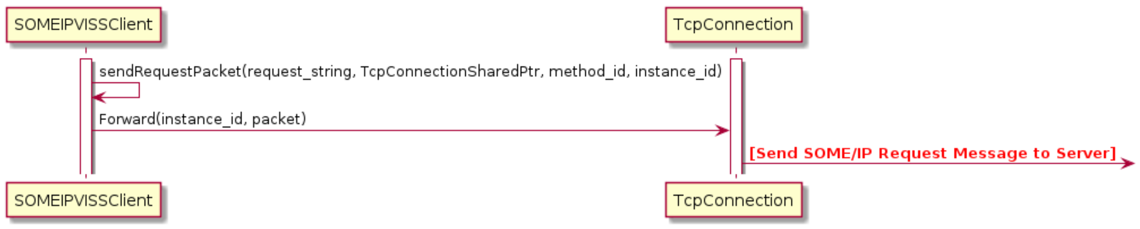

SOMEIPVISSClient
TcpConnection
sendRequestPacket(request_string, TcpConnectionSharedPtr, method_id, instance_id)
Forward(instance_id, packet)
[Send SOME/IP Request Message to Server]
SOMEIPVISSClient
TcpConnection

Figure 5.7: SOME/IP Request Message Transmission from Client

5.2.4 Reception of Request on Server

Figure 5.8 shows how a request message is received on the server from a client and processed. The TcpConnection object's HandleRead()and ProcessMessage() methods check the validity of the received request message. The process() method on ServerRequestProcessor is used to check and dispatch the requests to their corresponding handling sub functions. it does the following:

1. If the message type is 0x00 (Request), then do the following:
   a) If the method ID is 0x01, call the processVISSReadRequest(SomeIpMessage, ResponseSenderSharedPtr) method.
   b) Else if the method ID is 0x02, call the processVISSUpdateRequest(SomeIpMessage, ResponseSenderSharedPtr) method.
   c) Else if the method ID is 0x03, call the processVISSSubscribeRequest(SomeIpMessage, ResponseSenderSharedPtr) method.
   d) Else if the method ID is 0x04, call the processVISSUnsubscribeRequest(SomeIpMessage, ResponseSenderSharedPtr) method.
   e) Else, call processVISSError(SomeIpMessage, VISSErrorResponse(bad_request), ResponseSenderSharedPtr) method.

2. Else, call processVISSError(SomeIpMessage, VISSErrorResponse(bad_request), ResponseSenderSharedPtr) method.

Since, a subscribe and an unsubscribe requests do more then just sending a response. They need to be explained briefly. On the reception of subscribe request, the request is processed by the ProcessVISSSubscribeRequest() method in the following way:

1. Check the validity of the VISS subscribe request.
2. If valid request, then:
    a) Extract the path value from the request.
    b) If valid path, then:
        i. Assign a unique subscription ID and add the client's ResponseSenderSharedPtr along with the subscription ID to a list in the ServerSubscriptionMap[path].
        ii. Prepare positive subscribe response in response_string.
        iii. Call sendPositiveResponse(SomeIpMessage, response_string, ResponseSenderSharedPtr).

        Else, call processVISSError(SomeIpMessage, VISSErrorResponse(invalid_path), ResponseSenderSharedPtr).

    Else, call processVISSError(SomeIpMessage, VISSErrorResponse(bad_request), ResponseSenderSharedPtr).

On the reception of unsubscribe request, the request is processed by the ProcessVISSUnsubscribeRequest() method in the following way:

1. Check the validity of the VISS unsubscribe request.
2. If valid request, then:
    a) Extract the subscription ID from the request.
    b) Find the subscription ID in ServerSubscriptionMap by iterating over all keys and within all their respective lists.
    c) If a subscription is found with the specified subscription ID, then:
        i. Remove the subscription information from the list.
        ii. Prepare positive unsubscribe response in response_string.
        iii. call sendPositiveResponse(SomeIpMessage, response_string, ResponseSenderSharedPtr).
    d) Else, call call processVISSError(SomeIpMessage, VISSErrorResponse(invalid_subscription), ResponseSenderSharedPtr).
3. Else, call processVISSError(SomeIpMessage, VISSErrorResponse(bad_request), ResponseSenderSharedPtr).

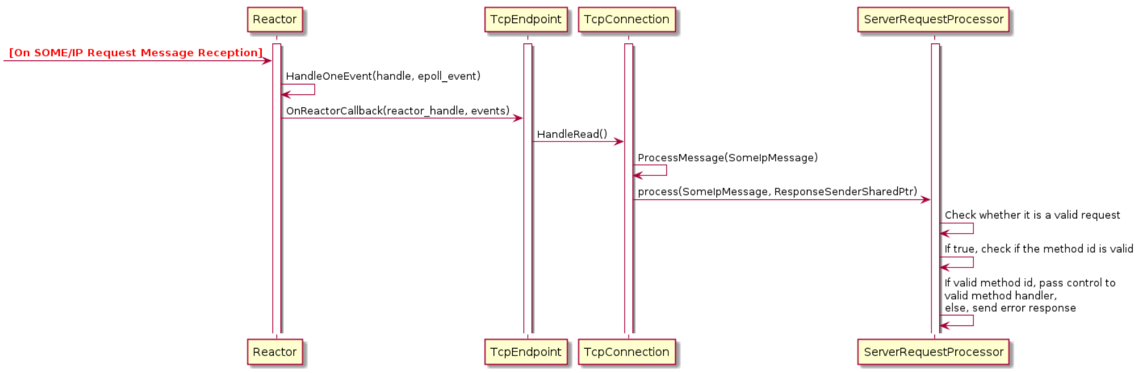

Reactor
TcpEndpoint
TcpConnection
ServerRequestProcessor
[On SOME/IP Request Message Reception]
HandleOneEvent(handle, epoll_event)
OnReactorCallback(reactor_handle, events)
HandleRead()
ProcessMessage(SomeIpMessage)
process(SomeIpMessage, ResponseSenderSharedPtr)
Check whether it is a valid request
If true, check if the method id is valid
If valid method id, pass control to valid method handler, else, send error response

Figure 5.8: SOME/IP Request Message Reception on Server

5.2.5 Sending Positive Response from Server

Figure 5.9 shows how a server sends a positive response to a client. sendPositiveResponsePacket() method prepares the SOME/IP positive response message and calls the TcpResponseSender's Forward() method for sending the response to the client. The sendPositiveResponsePacket() method does the following:

1. set the service ID, method ID, client ID, protocol version, interface version and session ID to the ones provided by the client.
2. Set the length field to 8 + length of the payload (which includes length field of response_string and response_string itself) (in bytes).
3. Set the message type to 0x80 (Positive Response).
4. Set the return code to 0x00 (OK).
5. Serialize the message and send it to server via TcpConnectionSharedPtr's Forward() method.

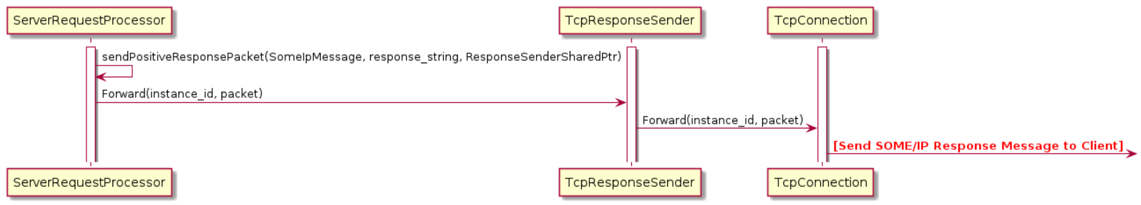

ServerRequestProcessor
TcpResponseSender
TcpConnection
sendPositiveResponsePacket(SomeIpMessage, response_string, ResponseSenderSharedPtr)
Forward(instance_id, packet)
Forward(instance_id, packet)
[Send SOME/IP Response Message to Client]

Figure 5.9: SOME/IP Positive Response Transmission from Server

5.2.6 Sending Error Response from Server

Figure 5.10 shows how a server sends an error response to a client. sendErrorResponsePacket() method prepares the SOME/IP error response message and calls the TcpResponseSender's Forward() method for sending the response to the client. The sendErrorResponsePacket() method does the following:

1. set the service ID, method ID, client ID, protocol version, interface version and session ID to the ones provided by the client.
2. Set the length field to 8 + length of the payload (which includes length field of response_string and response_string itself) (in bytes).
3. Set the message type to 0x81 (Error Response).
4. Set the return code to 0x01 (Not OK).
5. Serialize the message and send it to server via TcpConnectionSharedPtr's Forward() method.

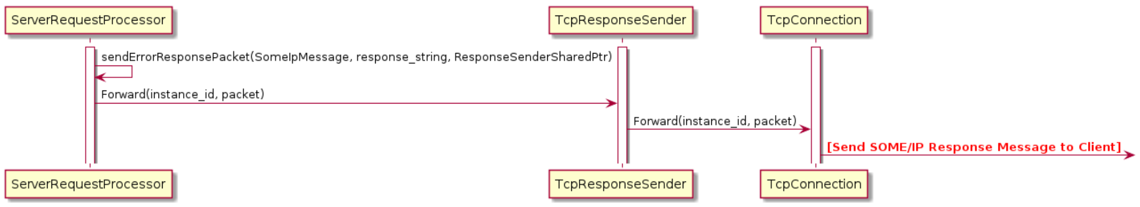

ServerRequestProcessor
TcpResponseSender
TcpConnection
sendErrorResponsePacket(SomeIpMessage, response_string, ResponseSenderSharedPtr)
Forward(instance_id, packet)
Forward(instance_id, packet)
[Send SOME/IP Response Message to Client]
ServerRequestProcessor
TcpResponseSender
TcpConnection

Figure 5.10: SOME/IP Negative Response Transmission from Server

5.2.7 Sending Notification from Server

Figure 5.11 shows how a server sends notification for a VISS subscription to a client. The publish() method is responsible for checking the subscribers who have subscribed to the signal and calling the sendNotificationPacket() method for the same. The sendNotificationPacket() is responsible for preparing the SOME/IP notification message and calling the TcpResponseSender's Forward() method for sending notification to the client. The sendNotificationPacket() method does the following:

1. Set the service ID to ServerConfig.service_id.
2. Set the method ID to 0x0005.
3. Set the client ID to 0.
4. Set the protocol version to 0x1.
5. Set the interface version to ServerConfig.major_version
6. Set the session ID to 0.
7. Set the length field to 8 + length of the payload (which includes length field of response_string and response_string itself) (in bytes).
8. Set the message type to 0x02 (Positive Response).
9. Set the return code to 0x00 (OK).
10. Serialize the message and send it to server via TcpConnectionSharedPtr's Forward() method.

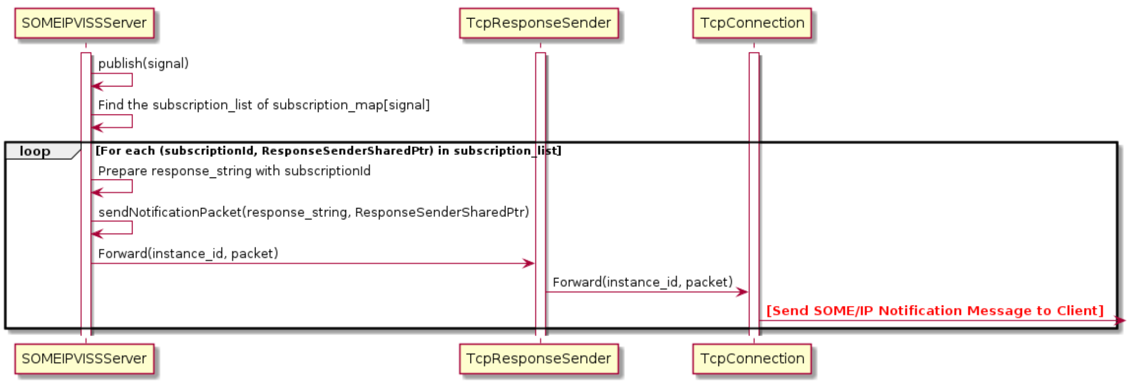

SOMEIPVISSServer
TcpResponseSender
TcpConnection
publish(signal)
Find the subscription_list of subscription_map[signal]
loop
[For each (subscriptionId, ResponseSenderSharedPtr) in subscription_list]
Prepare response_string with subscriptionId
sendNotificationPacket(response_string, ResponseSenderSharedPtr)
Forward(instance_id, packet)
Forward(instance_id, packet)
[Send SOME/IP Notification Message to Client]

Figure 5.11: SOME/IP Notification Transmission from Server

5.3 SOMEIPVISSClient API Flows

The SOMEIPVISSClient instance must first be initialized by passing a ClientConfig object in its constructor. After doing this, the various VISS methods can be called by passing JSON string corresponding to the request and each method call returns the response as a JSON string as well. This section first describes the private methods which act as helper functions and then the public methods which build up the client API.

5.3.1 Private Methods

discover

This method returns the server endpoint information and the instance ID, given the name of the signal to be found as input. It does the following:

1. Check if static service discovery is used (i.e. ClientConfig.signal_to_server_mapping contains the server providing the service for the specified signal).
2. If yes, then:
   a) Return the IP address, port and instance ID of the provided service instance.
3. Else:
   a) Prepare a SOME/IP-SD configuration option with the required signal (configuration_option).
   b) Initialize a ClientSDMessageProcessor object with service_id, instance_id, major_version, minor_version parameters from ClientConfig object and configuration_option.
   c) Initialize a ServiceDiscoveryEndpoint object with the Reactor, TimerManager, unicast_address, multicast_address and multicast_port from ClientConfig object.

d) Keep checking in a busy loop till ClientConfig..discovery_attempts seconds, if the ClientSDMessageProcessor object has found an endpoint.

e) If found within the time, then return the IP address, port and instance ID of the discovered service instance.

f) Else, if not found repeat steps in the busy loop.

g) If expiry reached, then return a null value indicating no service instance found.

find_connection

This method takes in the endpoint information (remote_endpoint) and returns a TcpConnection-SharedPtr. It is used to check if an existing open TCP connection between the client and a discovered server endpoint exists or not.

1. It checks it in the ClientSubscriptionMap object, iterating all keys until the value's endpoint information matches that of the provided one.
2. Else null object is returned, indicating no open TCP connection found.

connected

This method is used to check whether a connection is established within ClientConfig.connection_attempts seconds within a busy waiting loop given a TcpConnectionSharedPtr.

1. Return true, if the connection is established within the specified time.
2. Else, return false indicating no connection within the specified time was possible.

sendRequestPacket

This method is described in Section 5.2.3.

waitForResponse

This method performs a busy waiting till ClientConfig.response_attempts seconds.

1. If the response is received within the specified time, return true.
2. Else, return false.

5.3.2 Public Methods

Read

This method takes in a VISS JSON request string (request_string) as input and returns a VISS JSON response string as output. The input is a VISS read request and the output can be a read success response or an error response. Figure 5.12 shows the sequence diagram for a read method call.

Update

This method takes in a VISS JSON request string (request_string) as input and returns a VISS JSON response string as output. The input is a VISS update request and the output can be an update success response or an error response. Figure 5.13 shows the sequence diagram for an update method call.

Subscribe

This method takes in a VISS JSON request string (request_string) and a callback function (callback_fn) to be registered with the subscription notifications as input and returns a VISS JSON response string as output. The input is a VISS subscribe request and the output can be an subscribe success response or an error response. Figure 5.14 shows the sequence diagram for a subscribe method call.

Unsubscribe

This method takes in a VISS JSON request string (request_string) as input and returns a VISS JSON response string as output. The input is a VISS unsubscribe request and the output can be an unsubscribe success response or an error response. Figure 5.15 shows the sequence diagram for an unsubscribe method call.

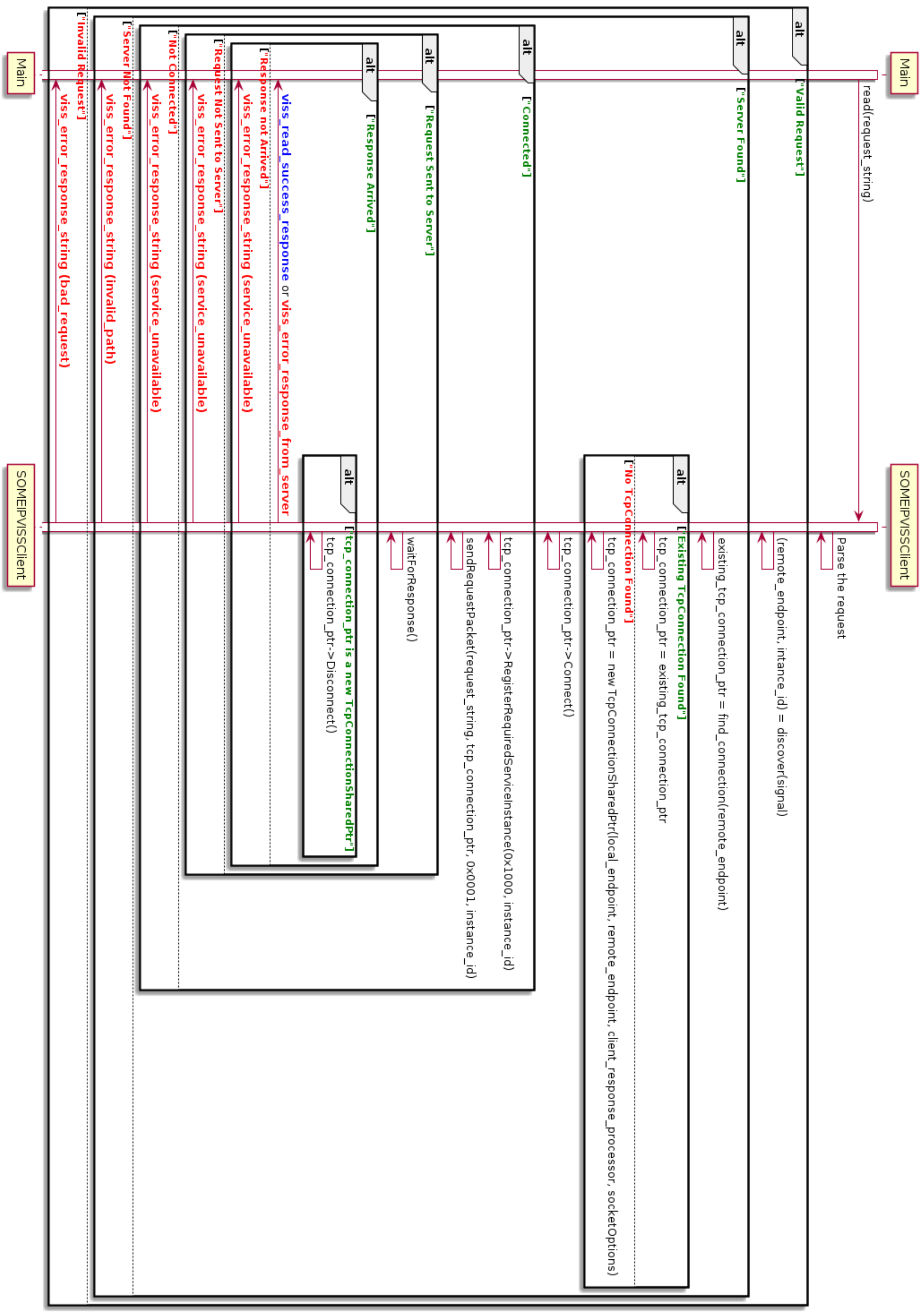

Main
SOMEIPVISSClient
read(request_string)
Parse the request
alt
["Valid Request"]
(remote_endpoint, intance_id) = discover(signal)
alt
["Server Found"]
existing_tcp_connection_ptr = find_connection(remote_endpoint)
alt
["Existing TcpConnection Found"]
tcp_connection_ptr = existing_tcp_connection_ptr
["No TcpConnection Found"]
tcp_connection_ptr = new TcpConnectionSharedPtr(local_endpoint, remote_endpoint, client_response_processor, socketOptions)
tcp_connection_ptr->Connect()
alt
["Connected"]
tcp_connection_ptr->RegisterRequiredServiceInstance(0x1000, instance_id)
sendRequestPacket(request_string, tcp_connection_ptr, 0x0001, instance_id)
alt
["Request Sent to Server"]
waitForResponse()
alt
["Response Arrived"]
alt
["tcp_connection_ptr is a new TcpConnectionSharedPtr"]
tcp_connection_ptr->Disconnect()
viss_read_success_response or viss_error_response_from_server
["Response not Arrived"]
viss_error_response_string (service_unavailable)
["Request Not Sent to Server"]
viss_error_response_string (service_unavailable)
["Not Connected"]
viss_error_response_string (service_unavailable)
["Server Not Found"]
viss_error_response_string (invalid_path)
["Invalid Request"]
viss_error_response_string (bad_request)
Main
SOMEIPVISSClient

Figure 5.12: SOMEIPVISSClient Read Flow

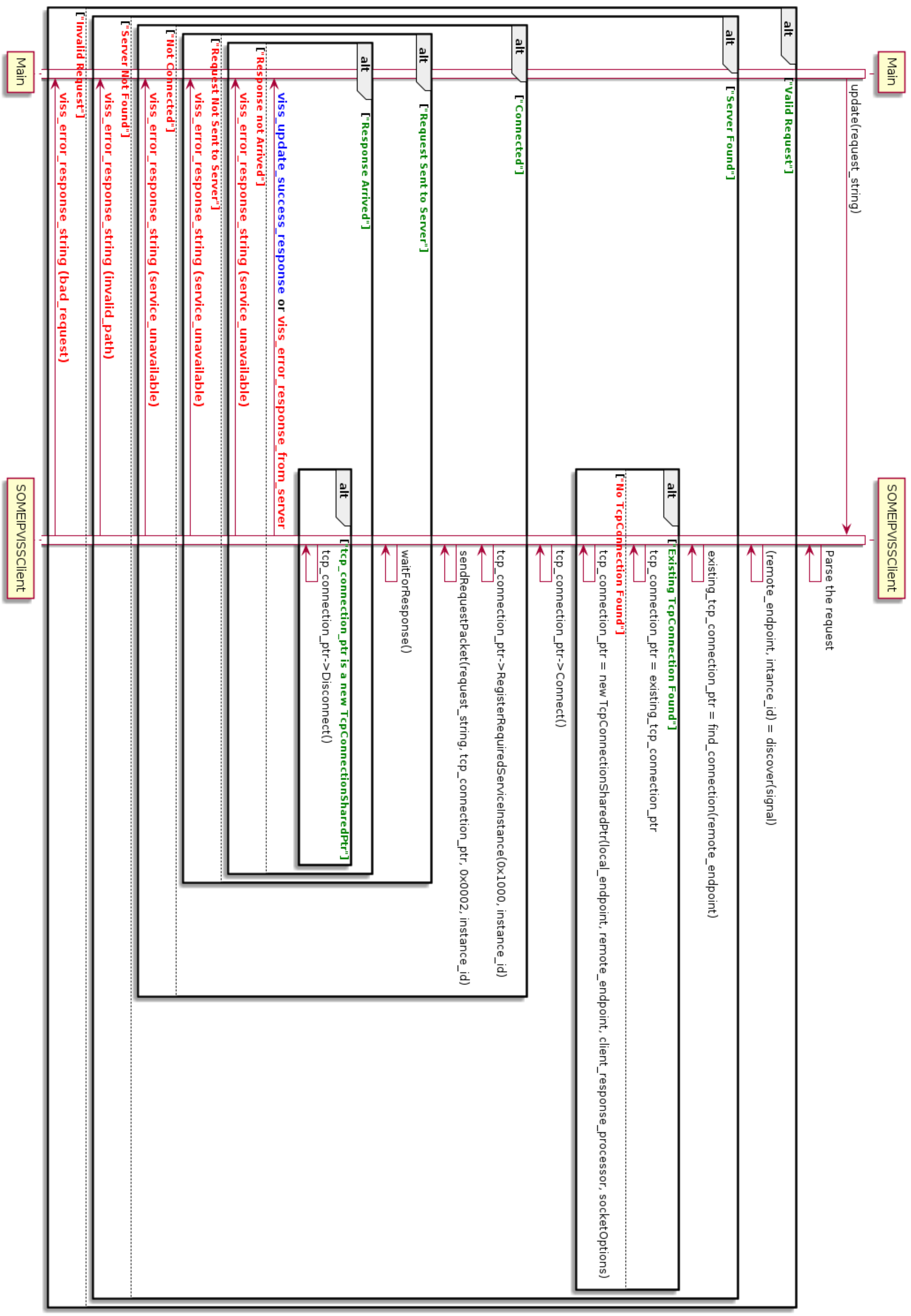

Main
SOMEIPVISSClient
update(request_string)
Parse the request
alt
["Valid Request"]
(remote_endpoint, intance_id) = discover(signal)
alt
["Server Found"]
existing_tcp_connection_ptr = find_connection(remote_endpoint)
alt
["Existing TcpConnection Found"]
tcp_connection_ptr = existing_tcp_connection_ptr
["No TcpConnection Found"]
tcp_connection_ptr = new TcpConnectionSharedPtr(local_endpoint, remote_endpoint, client_response_processor, socketOptions)
tcp_connection_ptr->Connect()
alt
["Connected"]
tcp_connection_ptr->RegisterRequiredServiceInstance(0x1000, instance_id)
sendRequestPacket(request_string, tcp_connection_ptr, 0x0002, instance_id)
alt
["Request Sent to Server"]
waitForResponse()
alt
["Response Arrived"]
alt
[tcp_connection_ptr is a new TcpConnectionSharedPtr"]
tcp_connection_ptr->Disconnect()
viss_update_success_response or viss_error_response_from_server
["Response not Arrived"]
viss_error_response_string (service_unavailable)
["Request Not Sent to Server"]
viss_error_response_string (service_unavailable)
["Not Connected"]
viss_error_response_string (service_unavailable)
["Server Not Found"]
viss_error_response_string (invalid_path)
["Invalid Request"]
viss_error_response_string (bad_request)
SOMEIPVISSClient
Main

Figure 5.13: SOMEIPVISSClient Update Flow

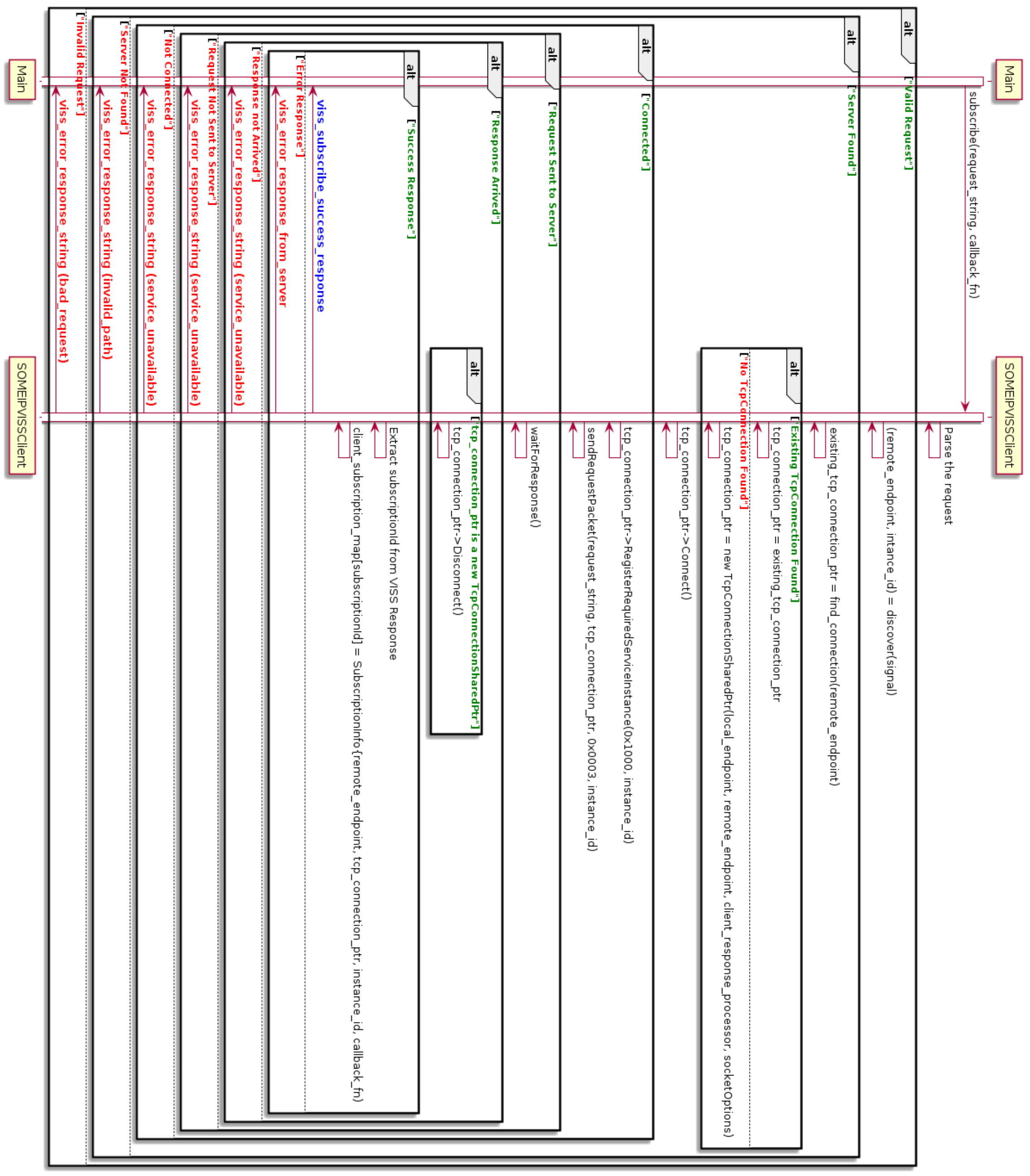

Main
SOMEIPVISSClient
subscribe(request_string, callback_fn)
Parse the request
alt
["Valid Request"]
(remote_endpoint, intance_id) = discover(signal)
alt
["Server Found"]
existing_tcp_connection_ptr = find_connection(remote_endpoint)
alt
["Existing TcpConnection Found"]
tcp_connection_ptr = existing_tcp_connection_ptr
["No TcpConnection Found"]
tcp_connection_ptr = new TcpConnectionSharedPtr(local_endpoint, remote_endpoint, client_response_processor, socketOptions)
tcp_connection_ptr->Connect()
alt
["Connected"]
tcp_connection_ptr->RegisterRequiredServiceInstance(0x1000, instance_id)
sendRequestPacket(request_string, tcp_connection_ptr, 0x0003, instance_id)
alt
["Request Sent to Server"]
waitForResponse()
alt
["Response Arrived"]
alt
[tcp_connection_ptr is a new TcpConnectionSharedPtr"]
tcp_connection_ptr->Disconnect()
alt
["Success Response"]
Extract subscriptionId from VISS Response
client_subscription_map[subscriptionId] = SubscriptionInfo{remote_endpoint, tcp_connection_ptr, instance_id, callback_fn}
viss_subscribe_success_response
["Error Response"]
viss_error_response_from_server
["Response not Arrived"]
viss_error_response_string (service_unavailable)
["Request Not Sent to Server"]
viss_error_response_string (service_unavailable)
["Not Connected"]
viss_error_response_string (service_unavailable)
["Server Not Found"]
viss_error_response_string (invalid_path)
["Invalid Request"]
viss_error_response_string (bad_request)

Figure 5.14: SOMEIPVISSClient Subscribe Flow

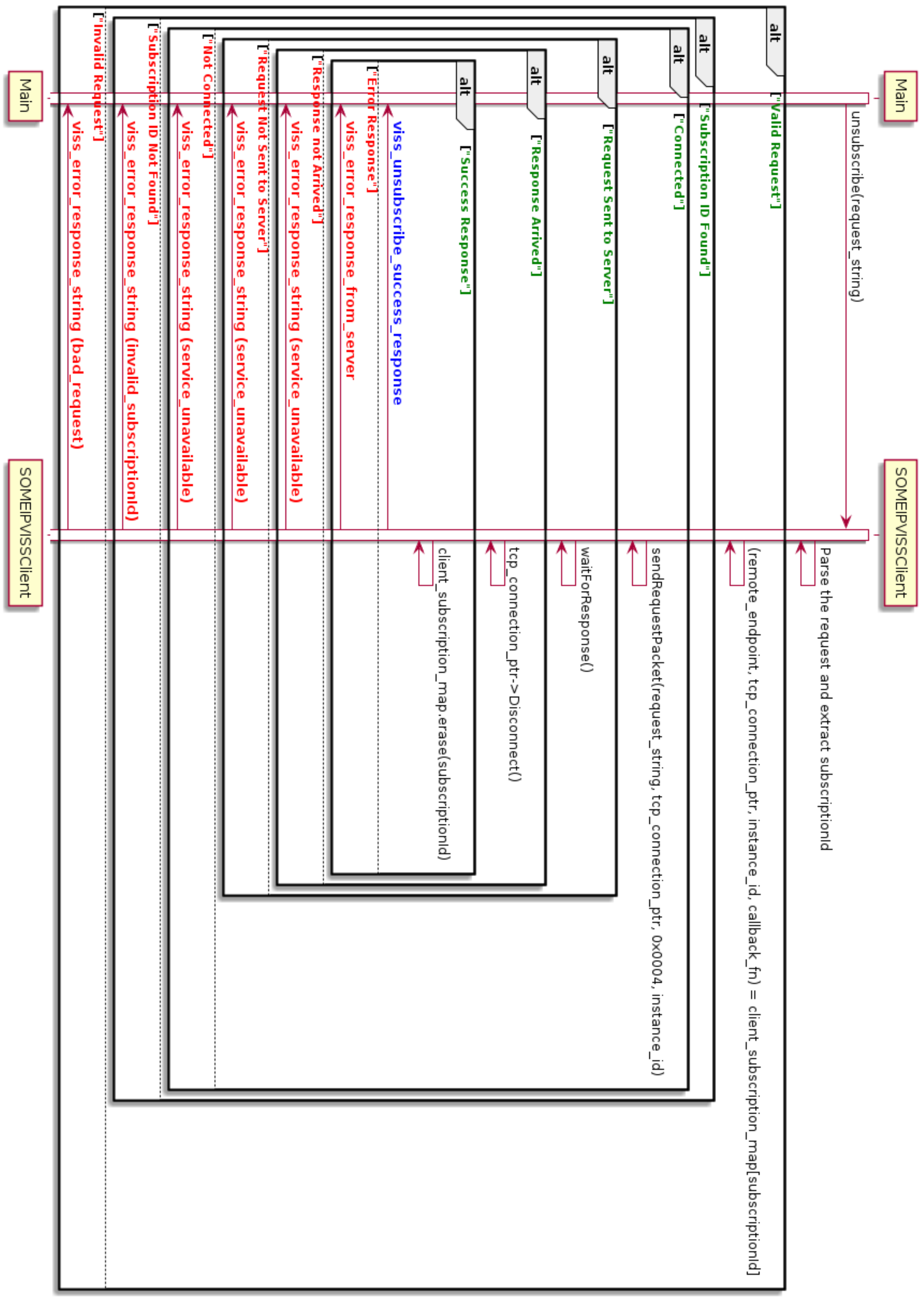

Main
SOMEIPVISSClient
unsubscribe(request_string)
alt
["Valid Request"]
Parse the request and extract subscriptionId
alt
["Subscription ID Found"]
(remote_endpoint, tcp_connection_ptr, instance_id, callback_fn) = client_subscription_map[subscriptionId]
alt
["Connected"]
sendRequestPacket(request_string, tcp_connection_ptr, 0x0004, instance_id)
alt
["Request Sent to Server"]
waitForResponse()
alt
["Response Arrived"]
tcp_connection_ptr->Disconnect()
alt
["Success Response"]
client_subscription_map.erase(subscriptionId)
viss_unsubscribe_success_response
["Error Response"]
viss_error_response_from_server
["Response not Arrived"]
viss_error_response_string (service_unavailable)
["Request Not Sent to Server"]
viss_error_response_string (service_unavailable)
["Not Connected"]
viss_error_response_string (service_unavailable)
["Subscription ID Not Found"]
viss_error_response_string (invalid_subscriptionId)
["Invalid Request"]
viss_error_response_string (bad_request)
Main
SOMEIPVISSClient

Figure 5.15: SOMEIPVISSClient Unsubscribe Flow

6 Evaluation

This chapter discusses the test scenario in Section 6.1, test setup in Section 6.2, measurements and some limitations associated with them in Section 6.3 and finally concludes with the results in Section 6.4.

6.1 Scenario

The test scenario consists of two SOME/IP based VISS servers and a single client. Server 1 contains the locking status for the left and right door for row 1 and Server 2 the same for row 2 (as shown in Figure 4.29). Both the servers publish the information about their left doors in case any client is interested to receive notifications about the status. Figure 6.1 shows the arrangement in an intuitive way). The proposed approaches using our implementation described in Section 4.7.1 and Section 4.7.2 are then evaluated for this test scenario for the JSON serialization scheme (Section 4.6.1). At the end, the maximum OfferService message size is tested and the results are described in Section 6.4.4.

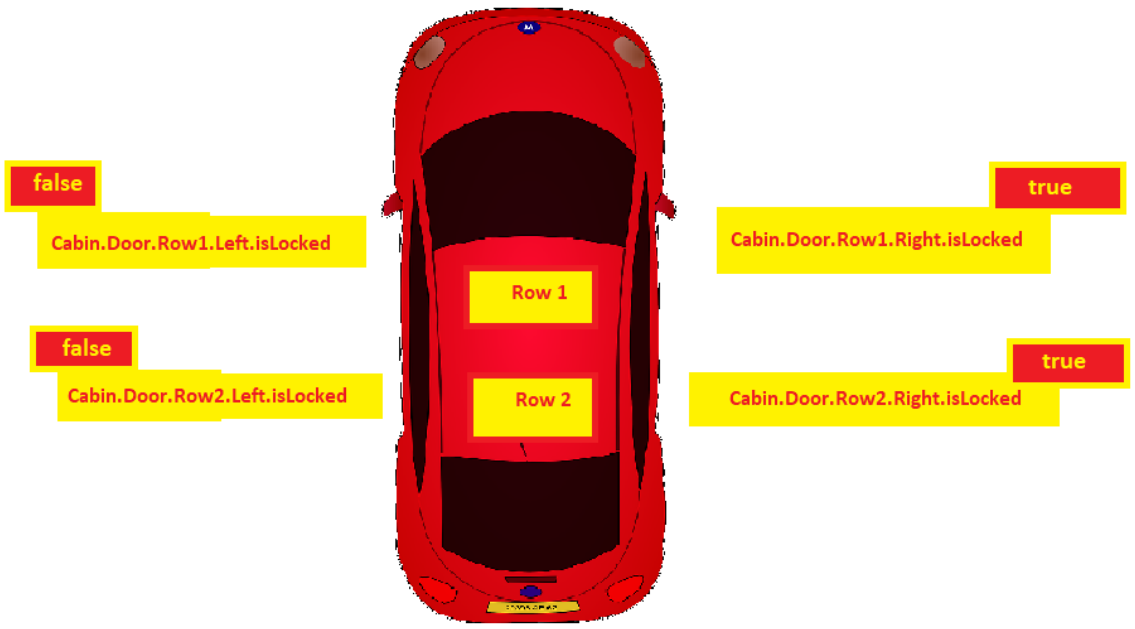

false
Cabin.Door.Row1.Left.isLocked
false
Cabin.Door.Row2.Left.isLocked
Row 1
Row 2
true
Cabin.Door.Row1.Right.isLocked
true
Cabin.Door.Row2.Right.isLocked

Figure 6.1: Model Car

6.2 Setup

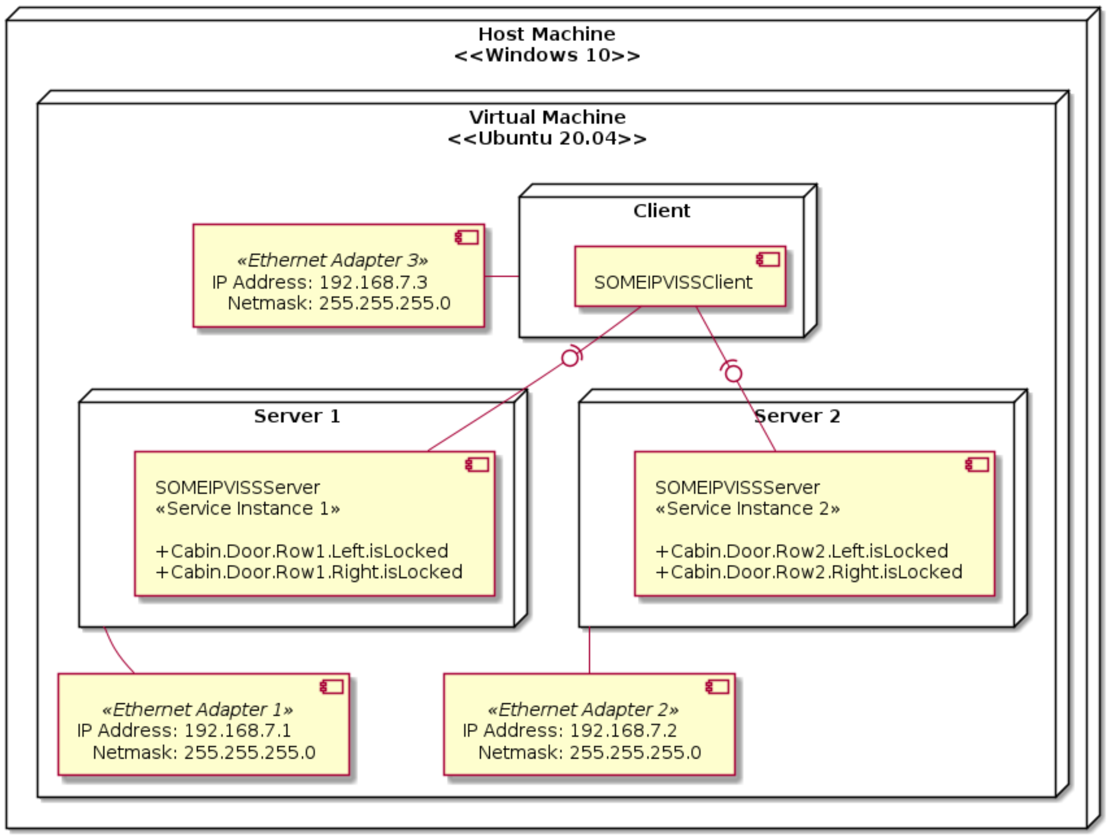


Figure 6.2: Deployment Diagram

The test scenario and further evaluations are performed on an Ubuntu 20.04 VM running on VMWare Workstation 15 Player. Figure 6.2 shows a deployment diagram with the setup, which consists of 3 virtual ethernet adapters each configured with it's own unique static IPv4 address and connected to Server 1, Server 2 and the client respectively. Each server process runs a single service instance and contains a set of mutually exclusive signals as described in previous section.

The general hardware specifications of the test machine are given in Table 6.1 and for the VM are given in Table 6.2.

OS	Windows 10
RAM	64 GB
CPU	Intel(R) Core(TM) i7-8850H CPU at 2.60GHz
GPU	NVIDIA Quadro P1000

Table 6.1: Host Machine Specifications

OS	Ubuntu 20.04
RAM	55.6 GB
Processors	12
Graphics Memory	768 MB
Hard Disk	300GB

Table 6.2: VM Specifications

The configurations for Server 1, Server 2 and the Client are given in Table 6.3, Table 6.4, and Table 6.5 respectively. The TCP and UDP ports are chosen in accordance with [7].

Unicast IP Address	192.168.7.1
Unicast Port	30501
Multicast IP Address	224.0.0.17
Multicast Port	30490
Service ID	0x1000
Instance ID	0x0001
Major Version	1
Minor Version	0
Offer Service TTL	3s
Offer Service Repetition Time	100ms
Signals	Cabin.Door.Row1.Left.isLocked, Cabin.Door.Row1.Right.isLocked

Table 6.3: Server 1 Configuration

Unicast IP Address	192.168.7.2
Unicast Port	30502
Multicast IP Address	224.0.0.17
Multicast Port	30490
Service ID	0x1000
Instance ID	0x0002
Major Version	1
Minor Version	0
Offer Service TTL	3s
Offer Service Repetition Time	100ms
Signals	Cabin.Door.Row2.Left.isLocked, Cabin.Door.Row2.Right.isLocked

Table 6.4: Server 2 Configuration

Unicast IP Address	192.168.7.3
Unicast Port	0 (Choose any Ephemeral port)
Multicast IP Address	224.0.0.17
Multicast Port	30490
Service ID	0x1000
Instance ID	0xffff
Major Version	1
Minor Version	0
Client ID	0x1234
Discovery attempts	5
Connection attempts	5
Response attempts	5

Table 6.5: Client Configuration

6.3 Measurements

Average response times and standard deviations for read, update, subscribe and unsubscribe requests as well as error responses are measured for all 4 possible signals using a test script. This is done for both dynamic and no service discovery cases. The test script calls the public methods of the SOMEIPVISSClient object and for each method various requests are sent. These requests are repeated for 10 times per set of methods to get more accurate average response times and standard deviations.

The test script also captures the packets in the sequence in a pcap file using dumpcap command line utility provided by Wireshark. These captures are used to analyze the average bandwidth requirements and packet structures of requests, responses, notifications and SOME/IP-SD OfferService messages.

Limitations regarding the fact that the ethernet adapters are virtual and the fact that it might take more time on VM for certain allocations and deallocations as compared to a physical machine, have allowed to test only 2 cases (no service discovery and a repetition time of 1s in case of dynamic service discovery). Some of these issues can be alleviated by perhaps using a smaller timescale for busy waiting and more mutexes. But this is a subject of optimization and due to constraints of time, have been left out of scope.

6.4 Results

This section describes the evaluations performed by capturing the packets and analyzing the pcap files for both the approaches described in Section 4.7.1 and Section 4.7.2. General statistics for both cases are first derived separately using Wireshark and then the the results are checked and insights are derived in Section 6.4.3. Finally the section is concluded with an additional test to check the implications when sending OfferService messages to the maximum allowed limit for a UDP message with IP fragmentation in Section 6.4.4.

6.4.1 No Service Discovery Case

For the approach described in Section 4.7.1, the execution of the test script took roughly 24 seconds, received 60 notifications and the rest of the results that it measured during the execution are presented in Table 6.6.

	Average Time	**Standard Deviation**
Success Read Response	5.033 ms	8.297 ms
Success Update Response	3.748 ms	0.435 ms
Success Subscribe Response	1.986 ms	0.513 ms
Success Unsubscribe Response	1.879 ms	0.207 ms
Error Response	40.948 ms	612.374 ms

Table 6.6: Response Statistics for No Service Discovery Case

Figure 6.3 shows the I/O graph for this approach. This graph indicates the throughput (or bandwidth requirements) during the execution in terms of bytes per second of packets transmitted between client and both the servers. Since there are no SOME/IP-SD messages present in this case, only SOME/IP messages and their corresponding TCP messages contribute to the bandwidth in this case. The average bandwidth is 5788.44 bytes per second according to the measurements, peaking at 65622 bytes per second.

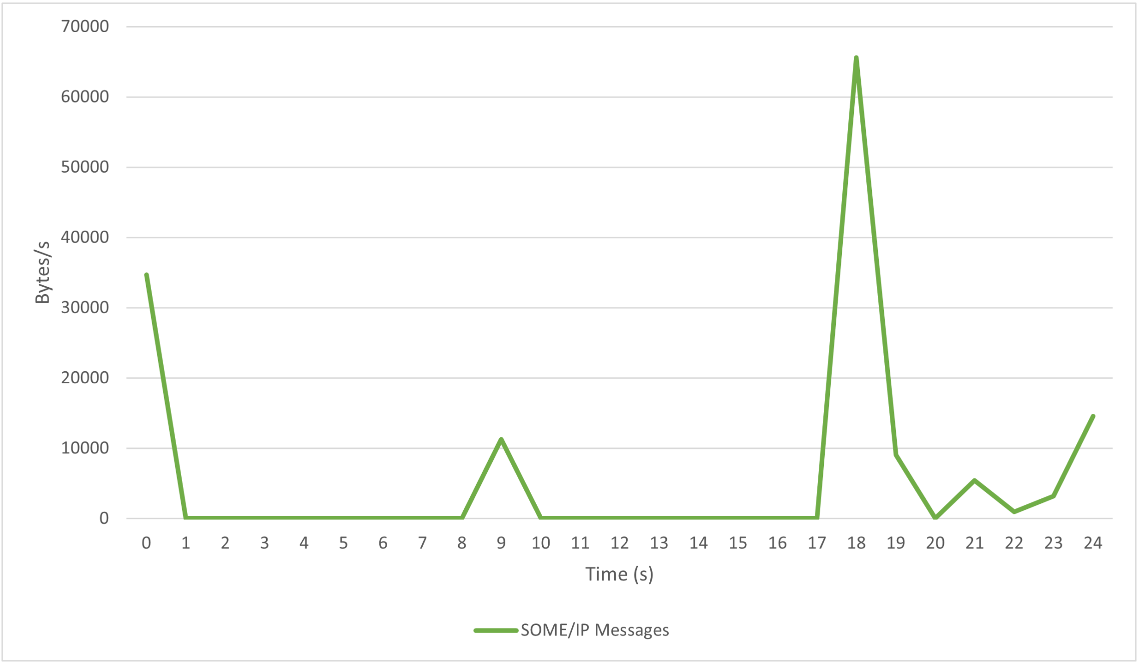
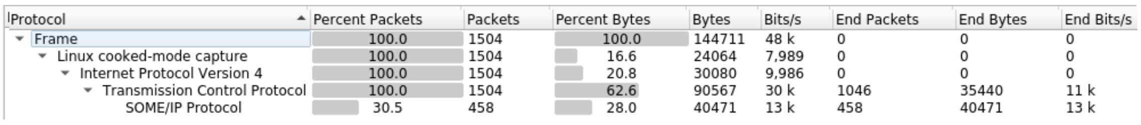


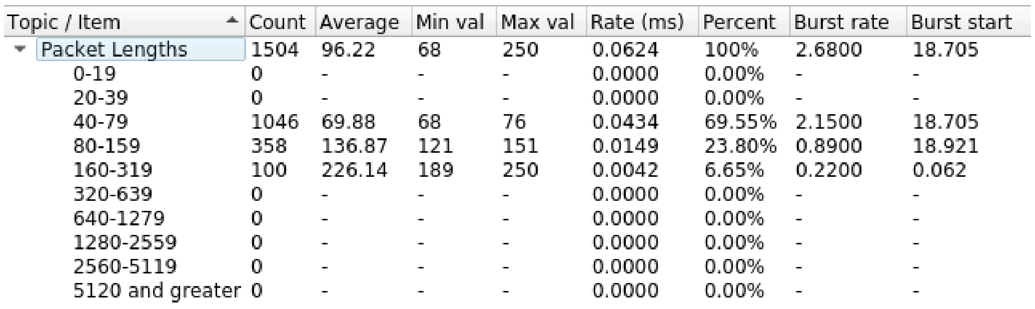
Figure 6.3: Bandwidth Usage in No Service Discovery Case

The Protocol Hierarchy and the Packet Lengths statistics via Wireshark are presented in Figure 6.4 and Figure 6.5 respectively. The Protocol Hierarchy shows the number of packets according to their protocol and the Packet Lengths shows the statistics of packets based on their lengths.



Protocol	Percent Packets	Packets	Percent Bytes	Bytes	Bits/s	End Packets	End Bytes	End Bits/s
Frame	100.0	1504	100.0	144711	48 k	0	0	0
Linux cooked-mode capture	100.0	1504	16.6	24064	7,989	0	0	0
Internet Protocol Version 4	100.0	1504	20.8	30080	9,986	0	0	0
Transmission Control Protocol	100.0	1504	62.6	90567	30 k	1046	35440	11 k
SOME/IP Protocol	30.5	458	28.0	40471	13 k	458	40471	13 k

Figure 6.4: Protocol Hierarchy in No Service Discovery Case



Topic / Item	Count	Average	Min val	Max val	Rate (ms)	Percent	Burst rate	Burst start
Packet Lengths	1504	96.22	68	250	0.0624	100%	2.6800	18.705
0-19	0	-	-	-	0.0000	0.00%	-	-
20-39	0	-	-	-	0.0000	0.00%	-	-
40-79	1046	69.88	68	76	0.0434	69.55%	2.1500	18.705
80-159	358	136.87	121	151	0.0149	23.80%	0.8900	18.921
160-319	100	226.14	189	250	0.0042	6.65%	0.2200	0.062
320-639	0	-	-	-	0.0000	0.00%	-	-
640-1279	0	-	-	-	0.0000	0.00%	-	-
1280-2559	0	-	-	-	0.0000	0.00%	-	-
2560-5119	0	-	-	-	0.0000	0.00%	-	-
5120 and greater	0	-	-	-	0.0000	0.00%	-	-

Figure 6.5: Packet Lengths in No Service Discovery Case

6.4.2 Dynamic Service Discovery Case

For the approach described in Section 4.7.2 with a repetition time of 1s, the execution of the test script took roughly 412 seconds, received 682 notifications and the rest of the results that it measured during the execution are presented in Table 6.7.

	Average Time	**Standard Deviation**
Success Read Response	714.557 ms	374.449 ms
Success Update Response	882.859 ms	300.228 ms
Success Subscribe Response	697.105 ms	383.849 ms
Success Unsubscribe Response	1.958 ms	0.161 ms
Error Response	623.994 ms	2273.99 ms

Table 6.7: Execution Statistics for Dynamic Service Discovery Case with Repetition Time = 1s

From the pcap file, the I/O graph, Protocol Hierarchy and the Packet Lengths statistics via Wireshark are presented in Figure 6.6 , Figure 6.7 and Figure 6.8 respectively.

For this case, The average bandwidths for SOME/IP-SD messages by Server 1 and Server 2 are 538.670 and 538.237 bytes per second respectively and for the SOME/IP messages is 845.600 bytes per second. This totals to a total average of 1922.507. The peak bandwidth of SOME/IP messages exceed that of SOME/IP-SD messages peaking at 11472 bytes per second.

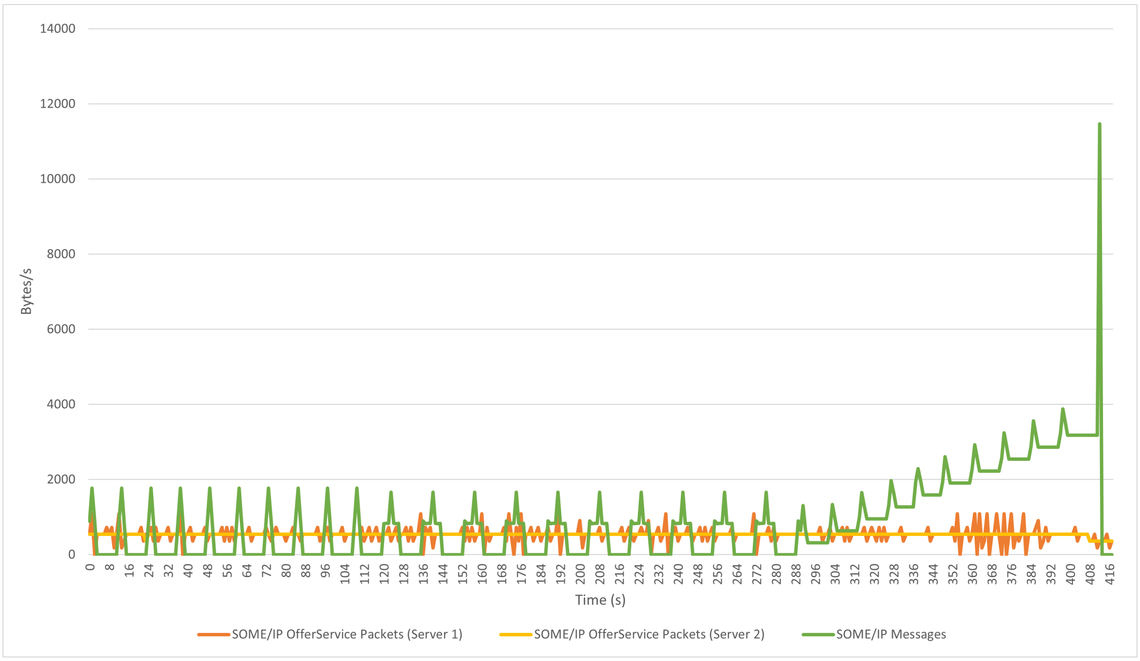
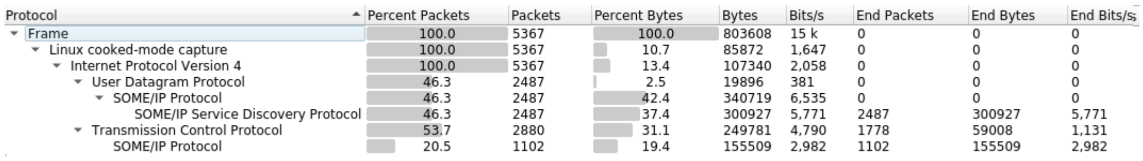
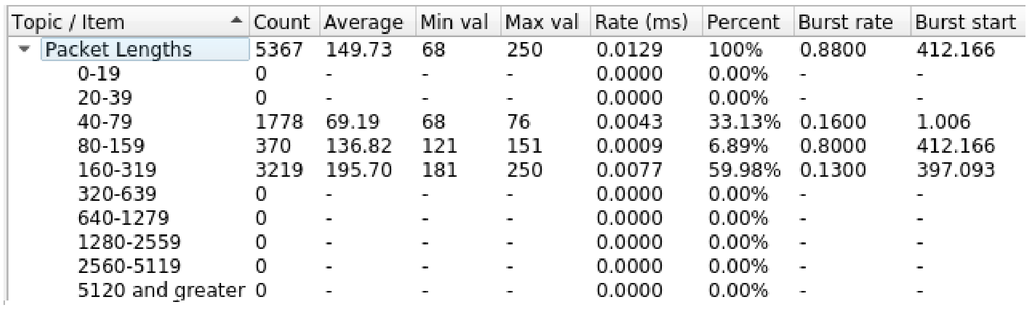


Figure 6.6: Bandwidth Usage in Dynamic Service Discovery Case with Repetition Time = 1s



Protocol	Percent Packets	Packets	Percent Bytes	Bytes	Bits/s	End Packets	End Bytes	End Bits/s
Frame	100.0	5367	100.0	803608	15 k	0	0	0
Linux cooked-mode capture	100.0	5367	10.7	85872	1,647	0	0	0
Internet Protocol Version 4	100.0	5367	13.4	107340	2,058	0	0	0
User Datagram Protocol	46.3	2487	2.5	19896	381	0	0	0
SOME/IP Protocol	46.3	2487	42.4	340719	6,535	0	0	0
SOME/IP Service Discovery Protocol	46.3	2487	37.4	300927	5,771	2487	300927	5,771
Transmission Control Protocol	53.7	2880	31.1	249781	4,790	1778	59008	1,131
SOME/IP Protocol	20.5	1102	19.4	155509	2,982	1102	155509	2,982

Figure 6.7: Protocol Hierarchy in Dynamic Service Discovery Case with Repetition Time = 1s



Topic / Item	Count	Average	Min val	Max val	Rate (ms)	Percent	Burst rate	Burst start
Packet Lengths	5367	149.73	68	250	0.0129	100%	0.8800	412.166
0-19	0	-	-	-	0.0000	0.00%	-	-
20-39	0	-	-	-	0.0000	0.00%	-	-
40-79	1778	69.19	68	76	0.0043	33.13%	0.1600	1.006
80-159	370	136.82	121	151	0.0009	6.89%	0.8000	412.166
160-319	3219	195.70	181	250	0.0077	59.98%	0.1300	397.093
320-639	0	-	-	-	0.0000	0.00%	-	-
640-1279	0	-	-	-	0.0000	0.00%	-	-
1280-2559	0	-	-	-	0.0000	0.00%	-	-
2560-5119	0	-	-	-	0.0000	0.00%	-	-
5120 and greater	0	-	-	-	0.0000	0.00%	-	-

Figure 6.8: Packet Lengths in Dynamic Service Discovery Case with Repetition Time = 1s

6.4.3 Insights on the Evaluation of Both the Cases

In Section 6.4.1 and Section 6.4.2, we see a huge difference in terms of round trip times for responses. The reason why the second approach takes more time currently is due to the fact that each OfferService message arrives after 1 second, and since the client only discovers the server when this message arrives, it has to wait in worst case for a maximum of about another 1 second for the next OfferService message to arrive. This is also the reason why the standard deviation is large, as the discovery can happen at any time in that 1 second interval.

The time required for unsubscribe responses to arrive remains same in both cases as unsubscribe requests rely on already opened connections which are kept track of by the client, so no service discovery needs to take place for sending an unsubscribe request, hence reducing the round trip time.

The bandwidth of approach 1 comes out to be larger as a result that packets are sent and received in a fraction of time as compared to approach 2. The transfer of packets via TCP for SOME/IP messages also increases the bandwidth as more messages including 3 way handshake and acknowledgements need to be sent additionally for these messages. UDP should decrease bandwidth due to the fact that there is less overhead as compared to TCP but at the cost of lesser reliability.

But this result does not conclude that approach 1 is better than approach 2, as the server addresses are not cached in these tests for the second approach, which would happen in a real case scenario. Also, to get a better overview and be able to compare both the approaches in a better way, more data is needed in terms of response time by varying repetition times, ideally with values less than 1 second like 100 millisecond, 1 millisecond, microseconds or nanoseconds, so that one might see the increasing bandwidth of SOME/IP-SD messages which would then add up to increase the bandwidth requirements of second approach as well. Due to certain limitations mentioned in Section 6.3, we were unable to test dynamic service discovery with repetition times below 1 second properly in our client. Also other parameters like CPU and RAM load might need to be tested preferably on a physical system rather than a VM to get a clearer picture.

To answer the question, why there are only 458 SOME/IP packets received in the first approach as compared to the 1102 in second approach, we looked at the protocol hierarchy of all possible SOME/IP messages to get a better picture. This shows that some requests were not sent by the client in the 1st approach which also lead to lesser responses. But the difference becomes huge due to the fact that the first approach receives at least 10 times less notifications, because it completes its execution in a fraction of time. Which is also contributed by faster subscribe responses which leads to unsubscribes to happen much earlier as compared to the second approach in our tests as the set of unsubscribe requests are sent after the completion of subscribe requests and responses. The expected and captured packets are also summarized in Table 6.8 in the order in which these requests are made (except OfferService and Notifications). The reason for loss of packets seems to be related to allocation of resources requiring more time to complete as compared to the requests themselves in some cases and is highly dependent on the state of VM, leading to connection failures between the client and the server.

Message Type	Expected Packets	Packets Captured In No SD Case	Packets Captured in Dynamic SD Case
Read Request	40	31	40
Read Response	40	31	40
Update Request	90	88	90
Update Response	80	79	80
Subscribe Request	40	40	40
Subscribe Response	40	40	40
Unsubscribe Request	40	40	40
Unsubscribe Response	40	40	40
Notifications	nil	60	682
Error Response	10	9	10
OfferService Messages	nil	0	2487

Table 6.8: Expected and Captured Packets for No and Dynamic Service Discovery Case

The messages for all types of requests, responses and notifications are captured in the pcap file. The serialization approach used is described in Section 4.6.1. This approach is used although being less efficient at storage compared to Section 4.6.2 as mentioned in Section 4.6.3 due to the fact that it is more closer to the VISS message format and is supported as UTF-8 string in SOME/IP as well. This makes it easier to analyze payload and evaluate the core concept.

Figure 6.9 and Figure 6.10 show the request response flows for the first and the second cases respectively for some VISS read requests.

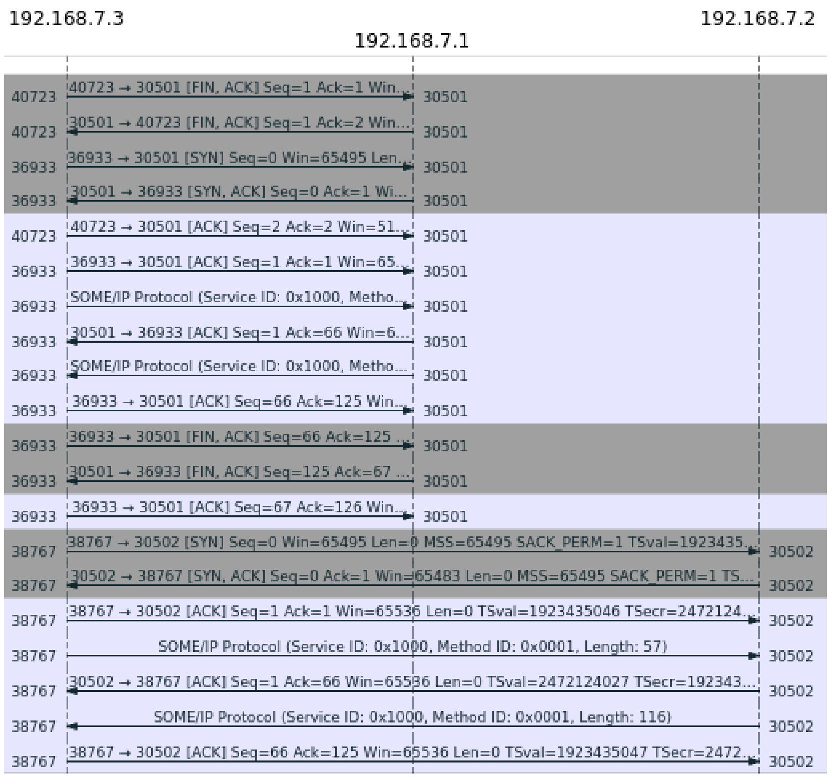
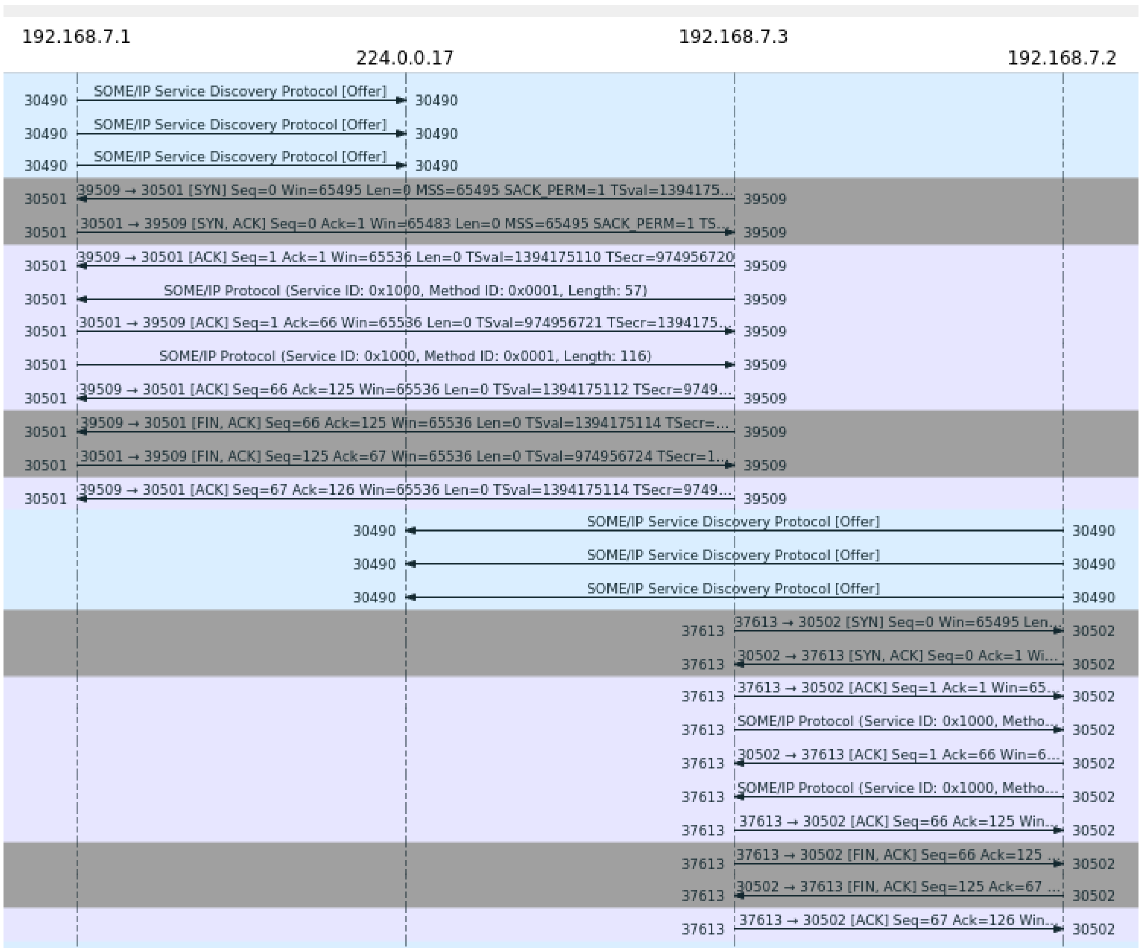

```
192.168.7.3                192.168.7.1                192.168.7.2
40723  40723 → 30501 [FIN, ACK] Seq=1 Ack=1 Win...  30501
40723  30501 → 40723 [FIN, ACK] Seq=1 Ack=2 Win...  30501
36933  36933 → 30501 [SYN] Seq=0 Win=65495 Len...  30501
36933  30501 → 36933 [SYN, ACK] Seq=0 Ack=1 Wi...  30501
40723  40723 → 30501 [ACK] Seq=2 Ack=2 Win=51...  30501
36933  36933 → 30501 [ACK] Seq=1 Ack=1 Win=65...  30501
36933  SOME/IP Protocol (Service ID: 0x1000, Metho...  30501
36933  30501 → 36933 [ACK] Seq=1 Ack=66 Win=6...  30501
36933  SOME/IP Protocol (Service ID: 0x1000, Metho...  30501
36933  36933 → 30501 [ACK] Seq=66 Ack=125 Win...  30501
36933  36933 → 30501 [FIN, ACK] Seq=66 Ack=125 ...  30501
36933  30501 → 36933 [FIN, ACK] Seq=125 Ack=67 ...  30501
36933  36933 → 30501 [ACK] Seq=67 Ack=126 Win...  30501
38767  38767 → 30502 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=1923435...  30502
38767  30502 → 38767 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TS...  30502
38767  38767 → 30502 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=1923435046 TSecr=2472124...  30502
38767  SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0001, Length: 57)  30502
38767  30502 → 38767 [ACK] Seq=1 Ack=66 Win=65536 Len=0 TSval=2472124027 TSecr=192343...  30502
38767  SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0001, Length: 116)  30502
38767  38767 → 30502 [ACK] Seq=66 Ack=125 Win=65536 Len=0 TSval=1923435047 TSecr=2472...  30502
```

Figure 6.9: Request / Response for No Service Discovery Case



```
192.168.7.1                    224.0.0.17                    192.168.7.3                    192.168.7.2
30490  SOME/IP Service Discovery Protocol [Offer]  30490
30490  SOME/IP Service Discovery Protocol [Offer]  30490
30490  SOME/IP Service Discovery Protocol [Offer]  30490
30501  39509 → 30501 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=1394175...  39509
30501  30501 → 39509 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TS...  39509
30501  39509 → 30501 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=1394175110 TSecr=974956720  39509
30501  SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0001, Length: 57)  39509
30501  30501 → 39509 [ACK] Seq=1 Ack=66 Win=65536 Len=0 TSval=974956721 TSecr=1394175...  39509
30501  SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0001, Length: 116)  39509
30501  39509 → 30501 [ACK] Seq=66 Ack=125 Win=65536 Len=0 TSval=1394175112 TSecr=9749...  39509
30501  39509 → 30501 [FIN, ACK] Seq=66 Ack=125 Win=65536 Len=0 TSval=1394175114 TSecr=...  39509
30501  30501 → 39509 [FIN, ACK] Seq=125 Ack=67 Win=65536 Len=0 TSval=974956724 TSecr=1...  39509
30501  39509 → 30501 [ACK] Seq=67 Ack=126 Win=65536 Len=0 TSval=1394175114 TSecr=9749...  39509
30490  SOME/IP Service Discovery Protocol [Offer]  30490
30490  SOME/IP Service Discovery Protocol [Offer]  30490
30490  SOME/IP Service Discovery Protocol [Offer]  30490
37613  37613 → 30502 [SYN] Seq=0 Win=65495 Len...  30502
37613  30502 → 37613 [SYN, ACK] Seq=0 Ack=1 Wi...  30502
37613  37613 → 30502 [ACK] Seq=1 Ack=1 Win=65...  30502
37613  SOME/IP Protocol (Service ID: 0x1000, Metho...  30502
37613  30502 → 37613 [ACK] Seq=1 Ack=66 Win=6...  30502
37613  SOME/IP Protocol (Service ID: 0x1000, Metho...  30502
37613  37613 → 30502 [ACK] Seq=66 Ack=125 Win...  30502
37613  37613 → 30502 [FIN, ACK] Seq=66 Ack=125 ...  30502
37613  30502 → 37613 [FIN, ACK] Seq=125 Ack=67 ...  30502
37613  37613 → 30502 [ACK] Seq=67 Ack=126 Win...  30502
```

Figure 6.10: Request / Response for Dynamic Service Discovery Case

Figure 6.11 and Figure 6.12 show the flow for VISS subscription messages as SOME/IP notifications. They are unidirectional and asynchronous (from server to client), but require an acknowledgement from client to server after each message due to TCP.

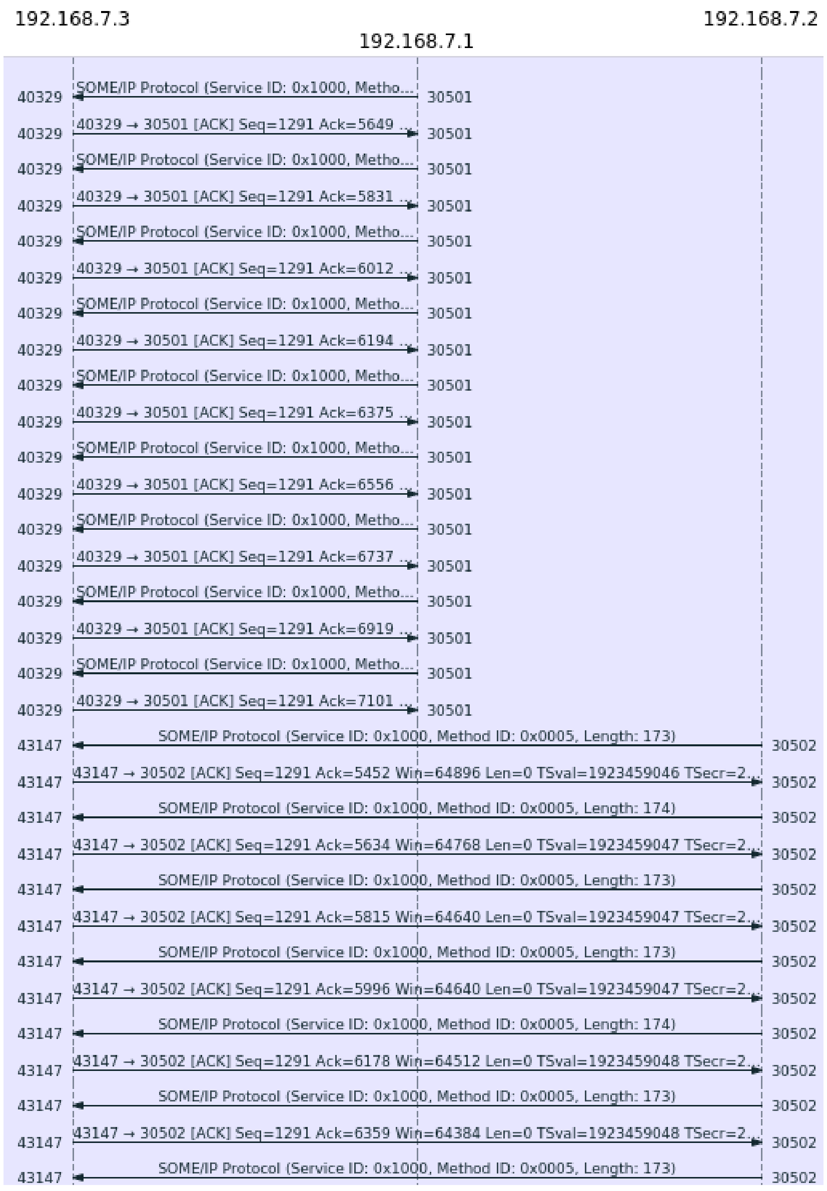

```
192.168.7.3                                                          192.168.7.2
                                192.168.7.1
40329  SOME/IP Protocol (Service ID: 0x1000, Metho...   30501
40329  40329 → 30501 [ACK] Seq=1291 Ack=5649 ...        30501
40329  SOME/IP Protocol (Service ID: 0x1000, Metho...   30501
40329  40329 → 30501 [ACK] Seq=1291 Ack=5831 ...        30501
40329  SOME/IP Protocol (Service ID: 0x1000, Metho...   30501
40329  40329 → 30501 [ACK] Seq=1291 Ack=6012 ...        30501
40329  SOME/IP Protocol (Service ID: 0x1000, Metho...   30501
40329  40329 → 30501 [ACK] Seq=1291 Ack=6194 ...        30501
40329  SOME/IP Protocol (Service ID: 0x1000, Metho...   30501
40329  40329 → 30501 [ACK] Seq=1291 Ack=6375 ...        30501
40329  SOME/IP Protocol (Service ID: 0x1000, Metho...   30501
40329  40329 → 30501 [ACK] Seq=1291 Ack=6556 ...        30501
40329  SOME/IP Protocol (Service ID: 0x1000, Metho...   30501
40329  40329 → 30501 [ACK] Seq=1291 Ack=6737 ...        30501
40329  SOME/IP Protocol (Service ID: 0x1000, Metho...   30501
40329  40329 → 30501 [ACK] Seq=1291 Ack=6919 ...        30501
40329  SOME/IP Protocol (Service ID: 0x1000, Metho...   30501
40329  40329 → 30501 [ACK] Seq=1291 Ack=7101 ...        30501
43147  SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0005, Length: 173)   30502
43147  43147 → 30502 [ACK] Seq=1291 Ack=5452 Win=64896 Len=0 TSval=1923459046 TSecr=2...   30502
43147  SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0005, Length: 174)   30502
43147  43147 → 30502 [ACK] Seq=1291 Ack=5634 Win=64768 Len=0 TSval=1923459047 TSecr=2...   30502
43147  SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0005, Length: 173)   30502
43147  43147 → 30502 [ACK] Seq=1291 Ack=5815 Win=64640 Len=0 TSval=1923459047 TSecr=2...   30502
43147  SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0005, Length: 173)   30502
43147  43147 → 30502 [ACK] Seq=1291 Ack=5996 Win=64640 Len=0 TSval=1923459047 TSecr=2...   30502
43147  SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0005, Length: 174)   30502
43147  43147 → 30502 [ACK] Seq=1291 Ack=6178 Win=64512 Len=0 TSval=1923459048 TSecr=2...   30502
43147  SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0005, Length: 173)   30502
43147  43147 → 30502 [ACK] Seq=1291 Ack=6359 Win=64384 Len=0 TSval=1923459048 TSecr=2...   30502
43147  SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0005, Length: 173)   30502
```

Figure 6.11: Notifications for No Service Discovery Case

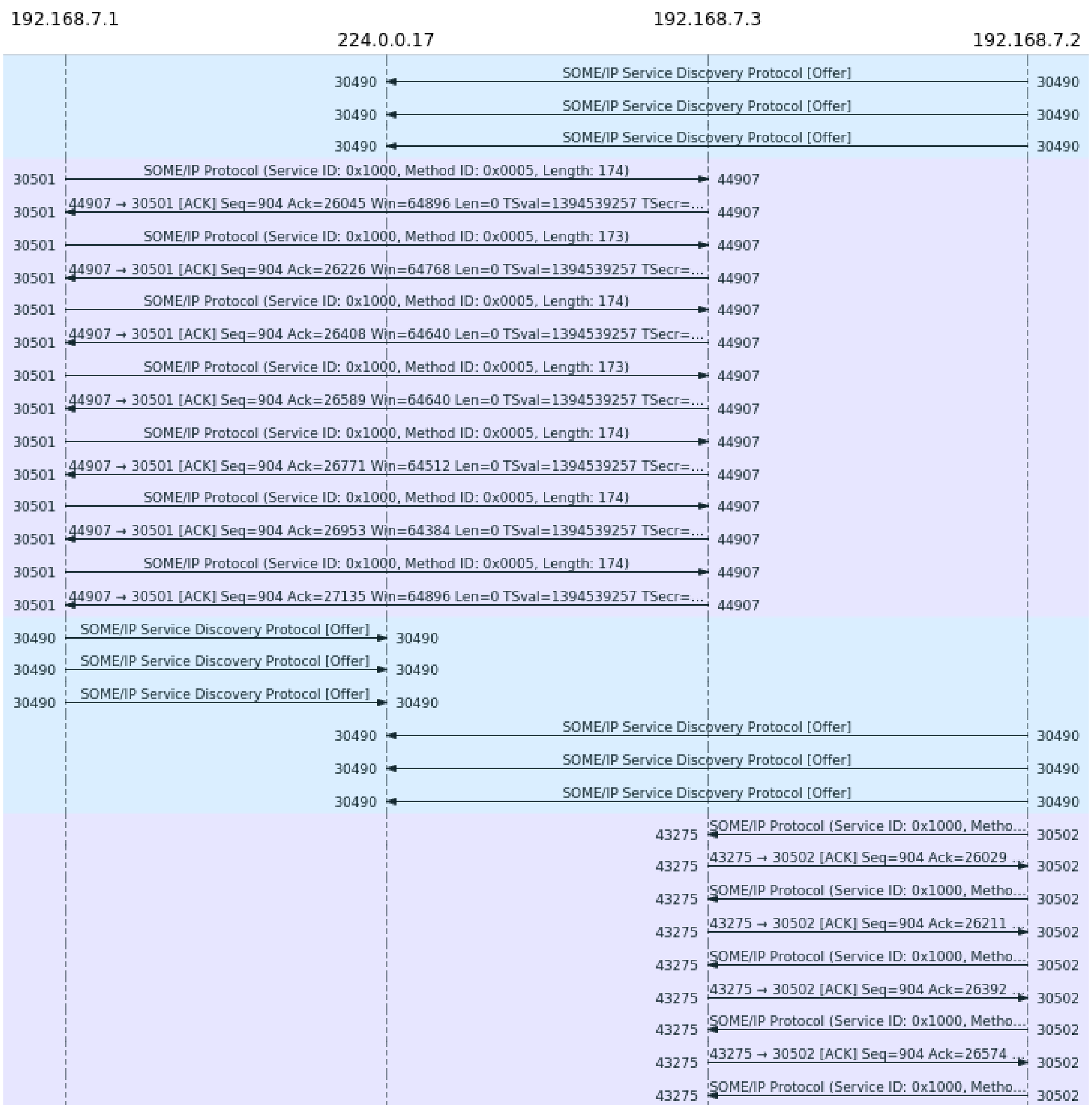
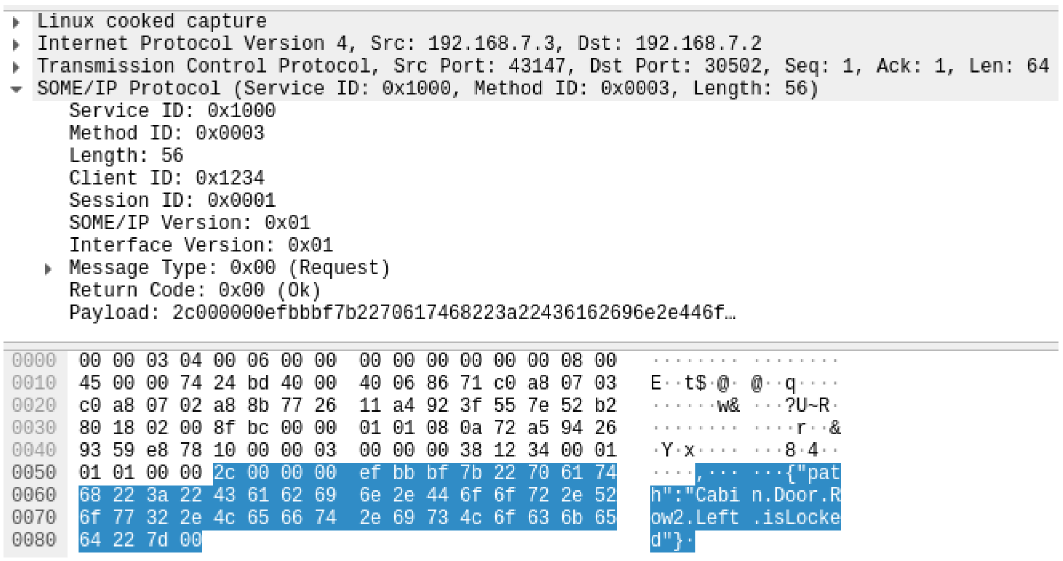

192.168.7.1
224.0.0.17
192.168.7.3
192.168.7.2
30490 ← SOME/IP Service Discovery Protocol [Offer] 30490
30490 ← SOME/IP Service Discovery Protocol [Offer] 30490
30490 ← SOME/IP Service Discovery Protocol [Offer] 30490
30501 SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0005, Length: 174) → 44907
30501 ← 44907 → 30501 [ACK] Seq=904 Ack=26045 Win=64896 Len=0 TSval=1394539257 TSecr=... 44907
30501 SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0005, Length: 173) → 44907
30501 ← 44907 → 30501 [ACK] Seq=904 Ack=26226 Win=64768 Len=0 TSval=1394539257 TSecr=... 44907
30501 SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0005, Length: 174) → 44907
30501 ← 44907 → 30501 [ACK] Seq=904 Ack=26408 Win=64640 Len=0 TSval=1394539257 TSecr=... 44907
30501 SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0005, Length: 173) → 44907
30501 ← 44907 → 30501 [ACK] Seq=904 Ack=26589 Win=64640 Len=0 TSval=1394539257 TSecr=... 44907
30501 SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0005, Length: 174) → 44907
30501 ← 44907 → 30501 [ACK] Seq=904 Ack=26771 Win=64512 Len=0 TSval=1394539257 TSecr=... 44907
30501 SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0005, Length: 174) → 44907
30501 ← 44907 → 30501 [ACK] Seq=904 Ack=26953 Win=64384 Len=0 TSval=1394539257 TSecr=... 44907
30501 SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0005, Length: 174) → 44907
30501 ← 44907 → 30501 [ACK] Seq=904 Ack=27135 Win=64896 Len=0 TSval=1394539257 TSecr=... 44907
30490 SOME/IP Service Discovery Protocol [Offer] → 30490
30490 SOME/IP Service Discovery Protocol [Offer] → 30490
30490 SOME/IP Service Discovery Protocol [Offer] → 30490
30490 ← SOME/IP Service Discovery Protocol [Offer] 30490
30490 ← SOME/IP Service Discovery Protocol [Offer] 30490
30490 ← SOME/IP Service Discovery Protocol [Offer] 30490
43275 ← SOME/IP Protocol (Service ID: 0x1000, Metho... 30502
43275 43275 → 30502 [ACK] Seq=904 Ack=26029 ... → 30502
43275 ← SOME/IP Protocol (Service ID: 0x1000, Metho... 30502
43275 43275 → 30502 [ACK] Seq=904 Ack=26211 ... → 30502
43275 ← SOME/IP Protocol (Service ID: 0x1000, Metho... 30502
43275 43275 → 30502 [ACK] Seq=904 Ack=26392 ... → 30502
43275 ← SOME/IP Protocol (Service ID: 0x1000, Metho... 30502
43275 43275 → 30502 [ACK] Seq=904 Ack=26574 ... → 30502
43275 ← SOME/IP Protocol (Service ID: 0x1000, Metho... 30502

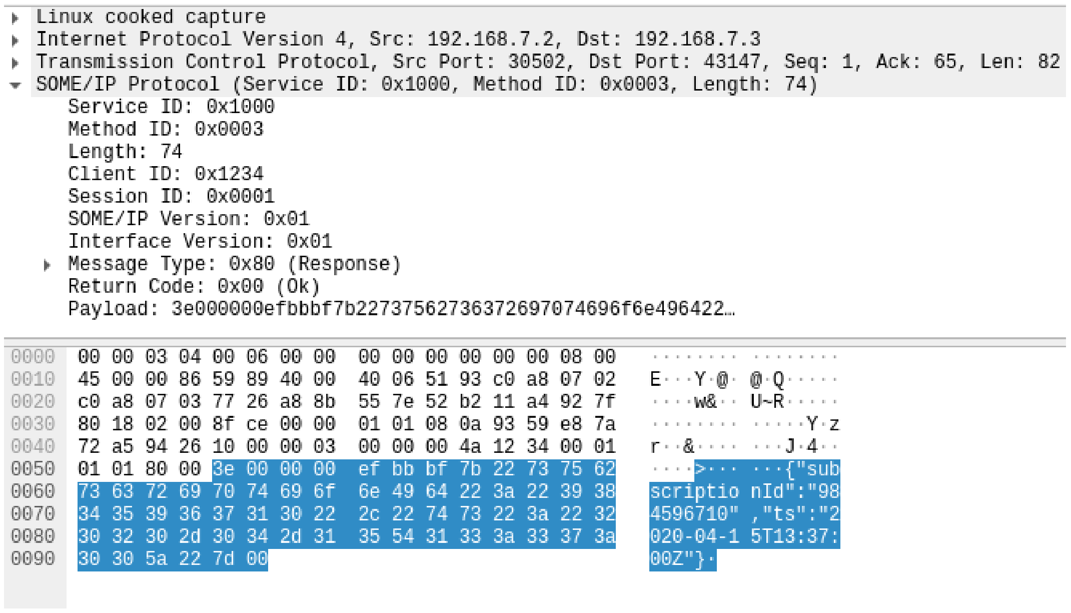
Figure 6.12: Notifications for Dynamic Service Discovery Case

Figure 6.13 and Figure 6.14 show the request and response payloads for a VISS subscribe request. The subscription ID returned in this case is 984596710 and the corresponding subscription notification for this ID is shown in Figure 6.15.



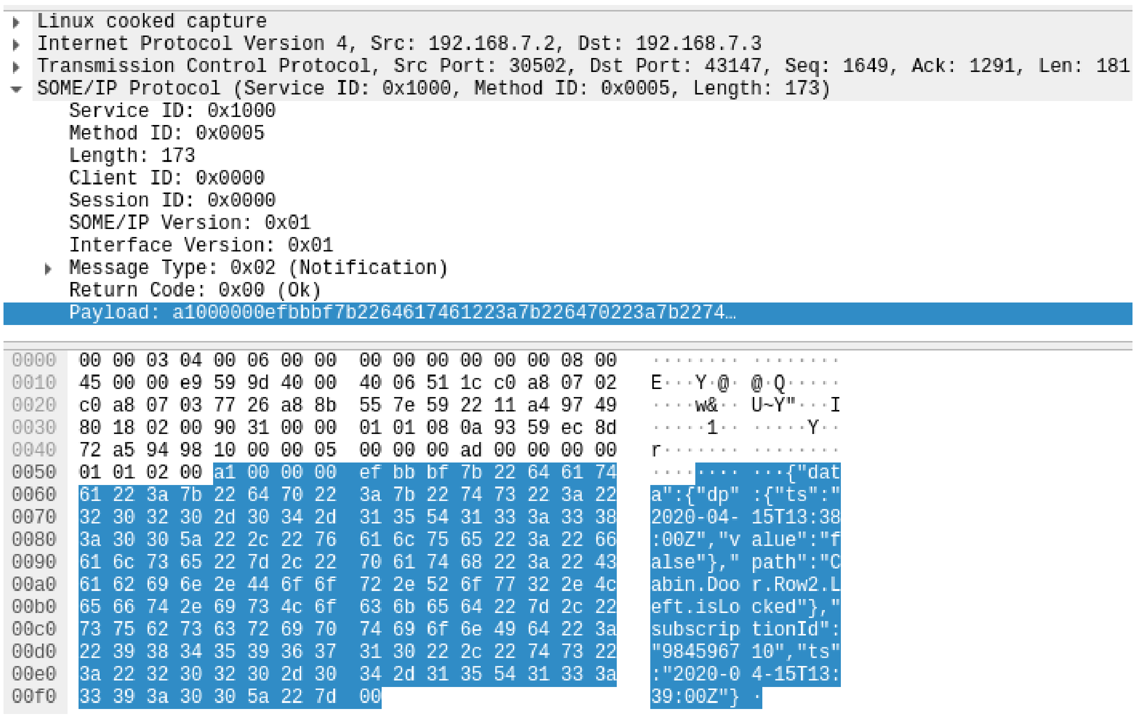
```
▸ Linux cooked capture
▸ Internet Protocol Version 4, Src: 192.168.7.3, Dst: 192.168.7.2
▸ Transmission Control Protocol, Src Port: 43147, Dst Port: 30502, Seq: 1, Ack: 1, Len: 64
▾ SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0003, Length: 56)
     Service ID: 0x1000
     Method ID: 0x0003
     Length: 56
     Client ID: 0x1234
     Session ID: 0x0001
     SOME/IP Version: 0x01
     Interface Version: 0x01
   ▸ Message Type: 0x00 (Request)
     Return Code: 0x00 (Ok)
     Payload: 2c000000efbbbf7b2270617468223a22436162696e2e446f…

0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 08 00   ········ ········
0010  45 00 00 74 24 bd 40 00  40 06 86 71 c0 a8 07 03   E··t$·@· @··q····
0020  c0 a8 07 02 a8 8b 77 26  11 a4 92 3f 55 7e 52 b2   ······w& ···?U~R·
0030  80 18 02 00 8f bc 00 00  01 01 08 0a 72 a5 94 26   ········ ····r··&
0040  93 59 e8 78 10 00 00 03  00 00 00 38 12 34 00 01   ·Y·x···· ···8·4··
0050  01 01 00 00 2c 00 00 00  ef bb bf 7b 22 70 61 74   ····,··· ···{"pat
0060  68 22 3a 22 43 61 62 69  6e 2e 44 6f 6f 72 2e 52   h":"Cabi n.Door.R
0070  6f 77 32 2e 4c 65 66 74  2e 69 73 4c 6f 63 6b 65   ow2.Left .isLocke
0080  64 22 7d 00                                        d"}·
```

Figure 6.13: Request Payload



```
▸ Linux cooked capture
▸ Internet Protocol Version 4, Src: 192.168.7.2, Dst: 192.168.7.3
▸ Transmission Control Protocol, Src Port: 30502, Dst Port: 43147, Seq: 1, Ack: 65, Len: 82
▾ SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0003, Length: 74)
     Service ID: 0x1000
     Method ID: 0x0003
     Length: 74
     Client ID: 0x1234
     Session ID: 0x0001
     SOME/IP Version: 0x01
     Interface Version: 0x01
   ▸ Message Type: 0x80 (Response)
     Return Code: 0x00 (Ok)
     Payload: 3e000000efbbbf7b22737562736372697074696f6e496422…

0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 08 00   ········ ········
0010  45 00 00 86 59 89 40 00  40 06 51 93 c0 a8 07 02   E···Y·@· @·Q·····
0020  c0 a8 07 03 77 26 a8 8b  55 7e 52 b2 11 a4 92 7f   ····w&·· U~R·····
0030  80 18 02 00 8f ce 00 00  01 01 08 0a 93 59 e8 7a   ········ ·····Y·z
0040  72 a5 94 26 10 00 00 03  00 00 00 4a 12 34 00 01   r··&···· ···J·4··
0050  01 01 80 00 3e 00 00 00  ef bb bf 7b 22 73 75 62   ····>··· ···{"sub
0060  73 63 72 69 70 74 69 6f  6e 49 64 22 3a 22 39 38   scriptio nId":"98
0070  34 35 39 36 37 31 30 22  2c 22 74 73 22 3a 22 32   4596710" ,"ts":"2
0080  30 32 30 2d 30 34 2d 31  35 54 31 33 3a 33 37 3a   020-04-1 5T13:37:
0090  30 30 5a 22 7d 00                                  00Z"}·
```

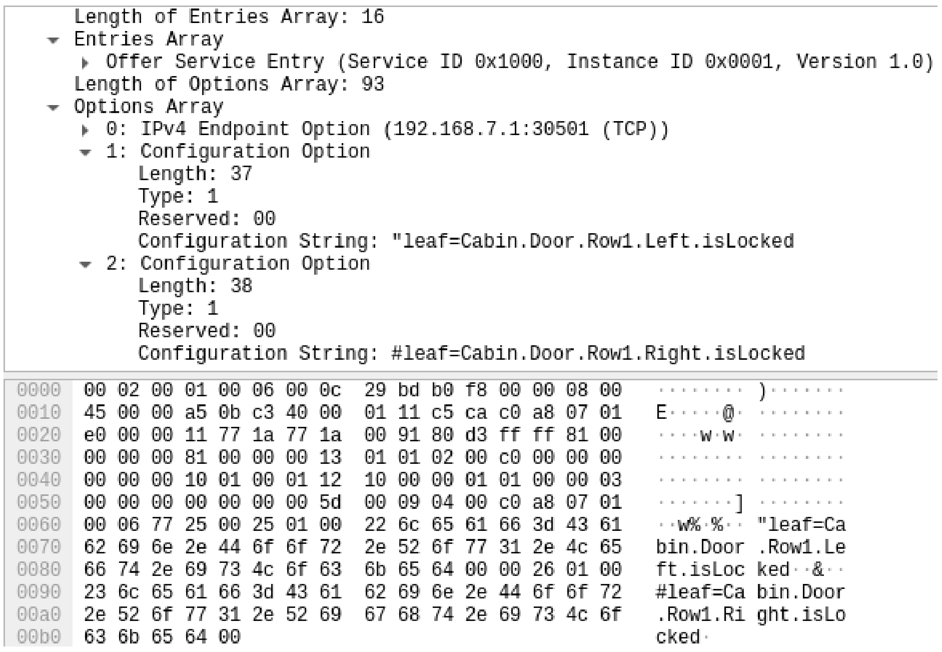
Figure 6.14: Response Payload



```
▸ Linux cooked capture
▸ Internet Protocol Version 4, Src: 192.168.7.2, Dst: 192.168.7.3
▸ Transmission Control Protocol, Src Port: 30502, Dst Port: 43147, Seq: 1649, Ack: 1291, Len: 181
▾ SOME/IP Protocol (Service ID: 0x1000, Method ID: 0x0005, Length: 173)
     Service ID: 0x1000
     Method ID: 0x0005
     Length: 173
     Client ID: 0x0000
     Session ID: 0x0000
     SOME/IP Version: 0x01
     Interface Version: 0x01
   ▸ Message Type: 0x02 (Notification)
     Return Code: 0x00 (Ok)
     Payload: a1000000efbbbf7b2264617461223a7b226470223a7b2274…

0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 08 00   ········ ········
0010  45 00 00 e9 59 9d 40 00  40 06 51 1c c0 a8 07 02   E···Y·@· @·Q·····
0020  c0 a8 07 03 77 26 a8 8b  55 7e 59 22 11 a4 97 49   ····w&·· U~Y"···I
0030  80 18 02 00 90 31 00 00  01 01 08 0a 93 59 ec 8d   ·····1·· ·····Y··
0040  72 a5 94 98 10 00 00 05  00 00 00 ad 00 00 00 00   r······· ········
0050  01 01 02 00 a1 00 00 00  ef bb bf 7b 22 64 61 74   ········ ···{"dat
0060  61 22 3a 7b 22 64 70 22  3a 7b 22 74 73 22 3a 22   a":{"dp" :{"ts":"
0070  32 30 32 30 2d 30 34 2d  31 35 54 31 33 3a 33 38   2020-04- 15T13:38
0080  3a 30 30 5a 22 2c 22 76  61 6c 75 65 22 3a 22 66   :00Z","v alue":"f
0090  61 6c 73 65 22 7d 2c 22  70 61 74 68 22 3a 22 43   alse"}," path":"C
00a0  61 62 69 6e 2e 44 6f 6f  72 2e 52 6f 77 32 2e 4c   abin.Doo r.Row2.L
00b0  65 66 74 2e 69 73 4c 6f  63 6b 65 64 22 7d 2c 22   eft.isLo cked"},"
00c0  73 75 62 73 63 72 69 70  74 69 6f 6e 49 64 22 3a   subscrip tionId":
00d0  22 39 38 34 35 39 36 37  31 30 22 2c 22 74 73 22   "9845967 10","ts"
00e0  3a 22 32 30 32 30 2d 30  34 2d 31 35 54 31 33 3a   :"2020-0 4-15T13:
00f0  33 39 3a 30 30 5a 22 7d  00                        39:00Z"} ·
```

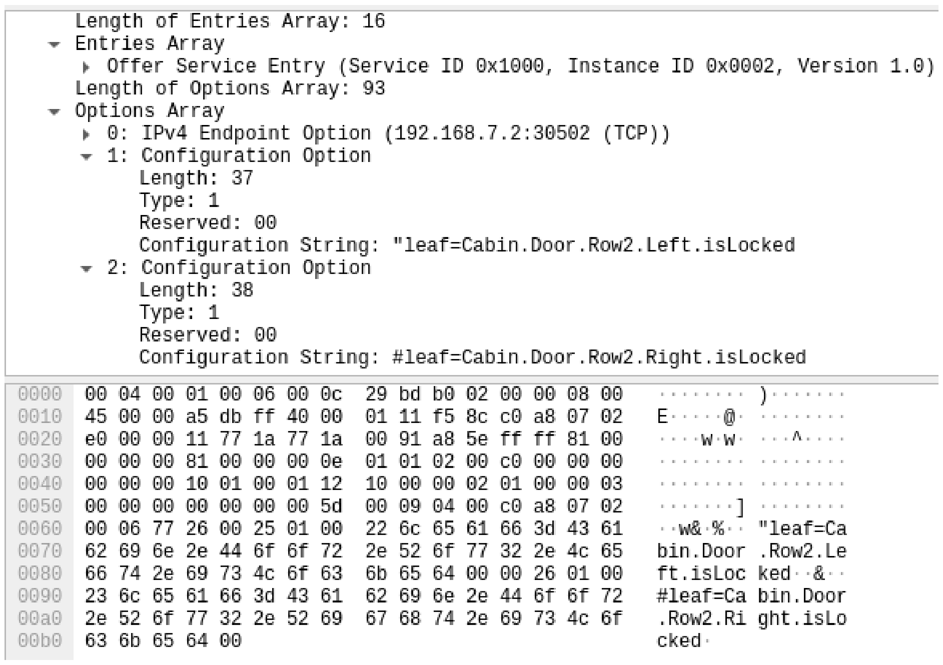
Figure 6.15: Subscription Payload

Figure 6.16 and Figure 6.17 show the SOME/IP-SD OfferService messages from server 1 and server 2 respectively containing the configuration options for signals handled by the respective servers.



```
     Length of Entries Array: 16
   ▾ Entries Array
       ▸ Offer Service Entry (Service ID 0x1000, Instance ID 0x0001, Version 1.0)
     Length of Options Array: 93
   ▾ Options Array
       ▸ 0: IPv4 Endpoint Option (192.168.7.1:30501 (TCP))
       ▾ 1: Configuration Option
              Length: 37
              Type: 1
              Reserved: 00
              Configuration String: "leaf=Cabin.Door.Row1.Left.isLocked
       ▾ 2: Configuration Option
              Length: 38
              Type: 1
              Reserved: 00
              Configuration String: #leaf=Cabin.Door.Row1.Right.isLocked

0000  00 02 00 01 00 06 00 0c  29 bd b0 f8 00 00 08 00   ········ )·······
0010  45 00 00 a5 0b c3 40 00  01 11 c5 ca c0 a8 07 01   E·····@· ········
0020  e0 00 00 11 77 1a 77 1a  00 91 80 d3 ff ff 81 00   ····w·w· ········
0030  00 00 00 81 00 00 00 13  01 01 02 00 c0 00 00 00   ········ ········
0040  00 00 00 10 01 00 01 12  10 00 00 01 01 00 00 03   ········ ········
0050  00 00 00 00 00 00 00 5d  00 09 04 00 c0 a8 07 01   ·······] ········
0060  00 06 77 25 00 25 01 00  22 6c 65 61 66 3d 43 61   ··w%·%·· "leaf=Ca
0070  62 69 6e 2e 44 6f 6f 72  2e 52 6f 77 31 2e 4c 65   bin.Door .Row1.Le
0080  66 74 2e 69 73 4c 6f 63  6b 65 64 00 00 26 01 00   ft.isLoc ked··&··
0090  23 6c 65 61 66 3d 43 61  62 69 6e 2e 44 6f 6f 72   #leaf=Ca bin.Door
00a0  2e 52 6f 77 31 2e 52 69  67 68 74 2e 69 73 4c 6f   .Row1.Ri ght.isLo
00b0  63 6b 65 64 00                                     cked·
```

Figure 6.16: OfferService Payload for Server 1



```
    Length of Entries Array: 16
  ▾ Entries Array
    ▸ Offer Service Entry (Service ID 0x1000, Instance ID 0x0002, Version 1.0)
    Length of Options Array: 93
  ▾ Options Array
    ▸ 0: IPv4 Endpoint Option (192.168.7.2:30502 (TCP))
    ▾ 1: Configuration Option
        Length: 37
        Type: 1
        Reserved: 00
        Configuration String: "leaf=Cabin.Door.Row2.Left.isLocked
    ▾ 2: Configuration Option
        Length: 38
        Type: 1
        Reserved: 00
        Configuration String: #leaf=Cabin.Door.Row2.Right.isLocked
```

```
0000  00 04 00 01 00 06 00 0c  29 bd b0 02 00 00 08 00   ········ )·······
0010  45 00 00 a5 db ff 40 00  01 11 f5 8c c0 a8 07 02   E····@· ········
0020  e0 00 00 11 77 1a 77 1a  00 91 a8 5e ff ff 81 00   ····w·w· ···^····
0030  00 00 00 81 00 00 00 0e  01 01 02 00 c0 00 00 00   ········ ········
0040  00 00 00 10 01 00 01 12  10 00 00 02 01 00 00 03   ········ ········
0050  00 00 00 00 00 00 00 5d  00 09 04 00 c0 a8 07 02   ·······] ········
0060  00 06 77 26 00 25 01 00  22 6c 65 61 66 3d 43 61   ··w&·%·· "leaf=Ca
0070  62 69 6e 2e 44 6f 6f 72  2e 52 6f 77 32 2e 4c 65   bin.Door .Row2.Le
0080  66 74 2e 69 73 4c 6f 63  6b 65 64 00 00 26 01 00   ft.isLoc ked··&··
0090  23 6c 65 61 66 3d 43 61  62 69 6e 2e 44 6f 6f 72   #leaf=Ca bin.Door
00a0  2e 52 6f 77 32 2e 52 69  67 68 74 2e 69 73 4c 6f   .Row2.Ri ght.isLo
00b0  63 6b 65 64 00                                     cked·
```

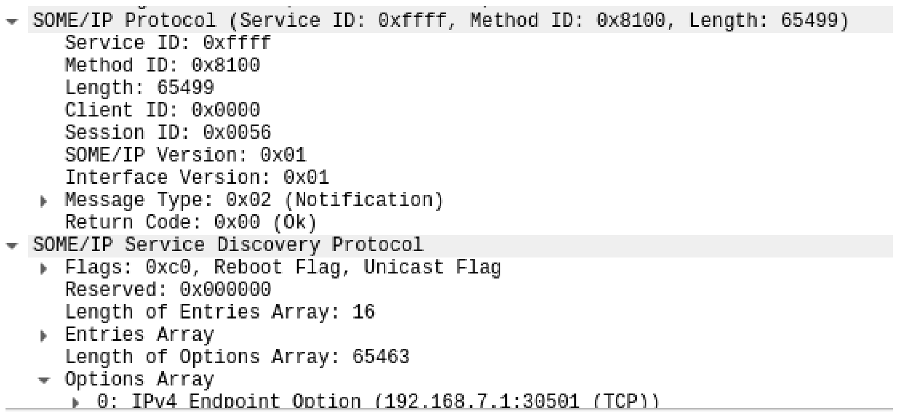
Figure 6.17: OfferService Payload for Server 2

6.4.4 Possible Amount of Signals via SOME/IP-SD Configuration Options

in Section 4.7.2, we discussed about the possibility of sending multiple configuration options. We have tried this approach by sending 1548 signals from Cabin.Door.Row0.Left.isLocked all the way up to Cabin.Door.Row1546.Left.isLocked and then a random string to make the SOME/IP-SD message length 65507 bytes at an interval of 1 millisecond, 10 milliseconds, 100 milliseconds and 1 second. Table 6.9. shows the average deltas for first 20 packets for all these cases. Figure 6.18 and Figure 6.19. But the period for OfferService messages becomes larger as these datagrams when fragmented via IP, need to be reassembled as well which takes time. A possibility to use SOME/IP-TP cannot be excluded as well. Main takeaway is that there is an effect on timing behaviour (mainly an increase in latency) of OfferService messages on increasing the number of signals.

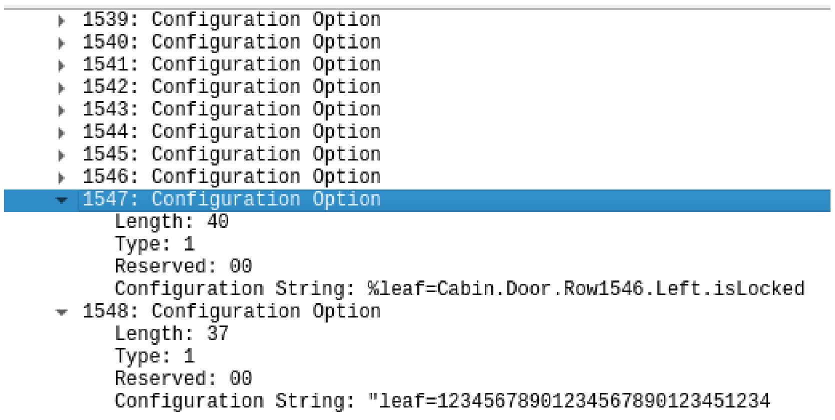
Repetition Time	Average Delta
1 ms	9.954795 ms
10 ms	19.731098 ms
100 ms	100.237863 ms
1 s	0.999907652 s

Table 6.9: Average Deltas for OfferService Messages Using Maximum Configuration Options Length



```
▾ SOME/IP Protocol (Service ID: 0xffff, Method ID: 0x8100, Length: 65499)
    Service ID: 0xffff
    Method ID: 0x8100
    Length: 65499
    Client ID: 0x0000
    Session ID: 0x0056
    SOME/IP Version: 0x01
    Interface Version: 0x01
  ▸ Message Type: 0x02 (Notification)
    Return Code: 0x00 (Ok)
▾ SOME/IP Service Discovery Protocol
  ▸ Flags: 0xc0, Reboot Flag, Unicast Flag
    Reserved: 0x000000
    Length of Entries Array: 16
  ▸ Entries Array
    Length of Options Array: 65463
  ▾ Options Array
    ▸ 0: IPv4 Endpoint Option (192.168.7.1:30501 (TCP))
```

Figure 6.18: OfferService Messages for Server 1 when message is fragmented



```
    ▸ 1539: Configuration Option
    ▸ 1540: Configuration Option
    ▸ 1541: Configuration Option
    ▸ 1542: Configuration Option
    ▸ 1543: Configuration Option
    ▸ 1544: Configuration Option
    ▸ 1545: Configuration Option
    ▸ 1546: Configuration Option
    ▾ 1547: Configuration Option
        Length: 40
        Type: 1
        Reserved: 00
        Configuration String: %leaf=Cabin.Door.Row1546.Left.isLocked
    ▾ 1548: Configuration Option
        Length: 37
        Type: 1
        Reserved: 00
        Configuration String: "leaf=123456789012345678901234512345
```

Figure 6.19: OfferService Messages for Server 1 when message is fragmented (Continued)

7 Conclusion and Outlook

7.1 Conclusion

We understood, designed, implemented and evaluated parts of the VISS server and client on SOME/IP. With the design and implementation, we went up and beyond and asked ourselves whether the SOME/IP stack in it's current state fits our scenario or not.

The answer to that question depends on the use case scenario. One can if using a centralized server scenario go with the all the approaches discussed in Section 4.2.1, Section 4.2.2 and Section 4.2.3, taking the trade-offs of all of them in account, or for that matter consider using already available transport binding, especially MQTT if the use case allows it.

The real value of SOME/IP comes in distributed server and service scenarios where the concept of service instances makes it a suitable candidate and also being a tried and tested solution in automotive environments, adds only more value to it. It seems promising as it allows flexibility with service discovery, allowing one to decide whether to do it dynamically or not do it at all by simply using a preconfigured signal to server mapping as described in Section 4.7.1. In case one decides to do it dynamically, one can go beyond the SOME/IP-SD standard and allow more configuration options to fit inside the OfferService message's referenced option array or fit multiple signals within a single configuration option, if one wants to follow the SOME/IP standard. Doing that paves way for approaches described Section 4.7.2 and Section 4.7.3.

As with everything currently available in the market, SOME/IP is also not an all-in-one or a perfect solution. if a few things are made more flexible, then it can become suitable for a range of solutions which were not imagined at it's design time. This includes offering more dedicated subscribe options than just eventgroups, like dedicated runtime subscription handling capabilities per client to allow for the dynamism needed to model subscription behavior for standards like VISS.

Although the results for service discovery tests described in Section 6.4.1 and Section 6.4.2 are non-conclusive. But what we did come up with, are the design aspects which needs to be taken care of, an experimental yet interesting implementation realizing not only the service discovery scenarios mentioned in Section 4.7.1 and Section 4.7.2, but also a dedicated application level subscription handling mechanism on both the client and the server, aspects which need to be tested, examples on how they can be tested and limiting factors along with room for improvement and extensions in the current SOME/IP standard.

7.2 Limitations

As stated in Section 6.3 and Section 6.4.3, the limitations posed by the VM and the fact that there needs to be more optimizations to be made like caching in the dynamic service discovery case, ask for more extensive testing. One then needs to evaluate the solution preferably on a real network of ECUs or physical hosts with repetition times under 1 second for the dynamic service discovery case. The fact that security and safety aspects need to be evaluated when deploying the solution on a real world vehicle cannot simply be neglected. By the end of this thesis, W3C released new draft specifications for both the core [1] and transport [50] of VISSv2. These standards introduce new changes including authorization token being an optional field in responses. We use the older specifications due to time constraints of the thesis. The server currently also does not check for duplicate subscriptions, which means if a client makes multiple subscribe requests for the same signal, each request is treated as a separate subscribe request with own unique subscription ID and hence the client can receive duplicate events if more than one subscribe request is made for the same signal without unsubscribing the other. We use separate configuration option per signal. This was done to make the implementation easier. But in a real world scenario, if one wants to follow the SOME/IP standard strictly, one can also put multiple signals inside a single configuration option as according to [10], it supports duplicate keys within different entries.

7.3 Future Work

The thesis could lead to the following future work:

1. Implementation and test using the proposed SOME/IP based serialization scheme (Section 4.6.2) and then comparing it practically with the one, which is used in the current implementation (Section 4.6.1).
2. Further optimizations to the current implementation like caching, checking for duplicate subscription on server, packing all the signals within a single configuration option (or perhaps finding a clever way of encoding it without mentioning each signal) and testing with repetition times under 1 second.
3. Implementation of the service discovery approach described in Section 4.7.3.
4. Although the subscription concept allows for expansion with rules for filters. It was not implemented. This needs to be seen in future along with how these filters will work with the distributed service scenario.
5. Security and rights management w.r.t onboard applications need to be researched on. Does one need the JWT based approach as described by [13] or other access management systems like, for example, Identity Access Management (IAM) [11] module on AUTOSAR Adaptive?
6. Error events have not been implemented (like token expired) as they heavily depend on the type of security measure used (in this case, the JWT token). These could be implemented in the future.

7. Middleware protocols like DDS look promising and COVESA also conducted a meeting to discuss the possibilities of VSS with DDS [19]. Hence, a detailed comparison of SOME/IP based implementation with the ones in DDS and MQTT could be another task.

Bibliography

[1] I. Agudo, U. Bjorkengren, W. Lee. *VISS version 2 - Core*. W3C Working Draft. https://www.w3.org/TR/2023/WD-viss2-core-20230927/. W3C, Sept. 2023 (cit. on p. 111).

[2] Amazon Web Services, Inc. *Reference Implementation for AWS IoT FleetWise*. URL: https://github.com/aws/aws-iot-fleetwise-edge (cit. on p. 40).

[3] Android Open Source Project. *Vehicle Hardware Abstraction Layer - Overview*. URL: https://source.android.com/docs/devices/automotive/vhal (cit. on p. 41).

[4] Android Open Source Project. *Vehicle Properties*. URL: https://source.android.com/docs/devices/automotive/vhal/properties (cit. on p. 41).

[5] S. Aust. "Vehicle API and Service Catalog for Next Generation Mobility". In: *2022 25th International Symposium on Wireless Personal Multimedia Communications (WPMC)*. 2022, pp. 418–423. DOI: 10.1109/WPMC55625.2022.10014905 (cit. on pp. 13, 16, 40).

[6] AutoReportNG. *7 Uses Of An API In The Automotive Industry*. 2022. URL: https://autoreportng.com/2022/10/7-uses-of-an-api-in-the-automotive-industry.html (cit. on p. 40).

[7] AUTOSAR. *Example for a Serialization Protocol (SOME/IP) V1.1.0 R4.1 Rev 3*. 637. Mar. 2014. URL: https://www.autosar.org/fileadmin/standards/R4-1/CP/AUTOSAR_TR_SomeIpExample.pdf (cit. on p. 96).

[8] AUTOSAR. *Methodology for Adaptive Platform, R22-11*. 709. Nov. 2022. URL: https://www.autosar.org/fileadmin/standards/R22-11/AP/AUTOSAR_TR_AdaptiveMethodology.pdf (cit. on p. 41).

[9] AUTOSAR. *SOME/IP Protocol Specification - Foundation R22-11*. 696. Nov. 2022. URL: https://www.autosar.org/fileadmin/standards/R22-11/FO/AUTOSAR_PRS_SOMEIPProtocol.pdf (cit. on pp. 13, 26–28, 33–39, 45, 46, 71).

[10] AUTOSAR. *SOME/IP Service Discovery Protocol Specification - Foundation R22-11*. 802. Nov. 2022. URL: https://www.autosar.org/fileadmin/standards/R22-11/FO/AUTOSAR_PRS_SOMEIPServiceDiscoveryProtocol.pdf (cit. on pp. 26, 28–33, 111).

[11] AUTOSAR. *Specification of Identity and Access Management AUTOSAR AP R22-11*. 900. Nov. 2022. URL: https://www.autosar.org/fileadmin/standards/R22-11/AP/AUTOSAR_SWS_IdentityAndAccessManagement.pdf (cit. on p. 111).

[12] Avnet, Inc. *Vehicle-to-everything (V2X) communication - the design engineer's guide*. 2023. URL: https://www.avnet.com/wps/portal/abacus/solutions/markets/automotive-and-transportation/automotive/communications-and-connectivity/v2x-communication/ (cit. on p. 13).

[13] U. Bjorkengren, W. Lee, I. Agudo. *VISS version 2 - Core*. W3C Working Draft. https://www.w3.org/TR/2023/WD-viss2-core-20230418/. W3C, Apr. 2023 (cit. on pp. 13, 19, 20, 44, 66, 111).

[14] BMW Group. *BMW Open Data Platform*. 2020. URL: https://bmw-cardata.bmwgroup.com/thirdparty/public/car-data (cit. on p. 40).

[15] R. T. Braden. *Requirements for Internet Hosts - Application and Support*. RFC 1123. Oct. 1989. DOI: 10.17487/RFC1123. URL: https://www.rfc-editor.org/info/rfc1123 (cit. on p. 26).

[16] R. T. Braden. *Requirements for Internet Hosts - Communication Layers*. RFC 1122. Oct. 1989. DOI: 10.17487/RFC1122. URL: https://www.rfc-editor.org/info/rfc1122 (cit. on p. 26).

[17] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. Dec. 2017. DOI: 10.17487/RFC8259. URL: https://www.rfc-editor.org/info/rfc8259 (cit. on pp. 19, 46).

[18] COVESA. *Basic Rules :: Vehicle Signal Specification*. URL: https://covesa.github.io/vehicle_signal_specification/rule_set/basics/#specification-format (cit. on p. 18).

[19] COVESA. *COVESA Spotlight: How DDS Enhances VSS for Software Defined Vehicle - YouTube*. URL: https://www.youtube.com/watch?v=2NRhBlZXG_0 (cit. on p. 112).

[20] COVESA. *Includes :: Vehicle Signal Specification*. URL: https://covesa.github.io/vehicle_signal_specification/rule_set/includes/ (cit. on p. 18).

[21] COVESA. *Overlay :: Vehicle Signal Specification*. URL: https://covesa.github.io/vehicle_signal_specification/rule_set/overlay/ (cit. on p. 18).

[22] COVESA. *Vehicle API - Wiki Front Page*. 2022. URL: https://wiki.covesa.global/display/WIK4/Vehicle+API (cit. on p. 16).

[23] COVESA. *Vehicle Service Catalog (VSC) - Common Interface Description Model*. 2022. URL: https://wiki.covesa.global/display/WIK4/Vehicle+Service+Catalog+%28VSC%29+-+Common+Interface+Description+Model (cit. on pp. 16, 40).

[24] COVESA. *Vehicle Service Catalog (VSC) GitHub Repository*. 2022. URL: https://github.com/COVESA/vehicle_service_catalog (cit. on p. 16).

[25] COVESA. *Vehicle Signal Specification*. URL: https://covesa.github.io/vehicle_signal_specification/ (cit. on pp. 13, 17).

[26] COVESA. *VSS Resources at a Glance - Wiki Front Page*. URL: https://wiki.covesa.global/display/WIK4/VSS+Resources+at+a+Glance (cit. on p. 40).

[27] COVESA. *vss-tools GitHub Repository*. URL: https://github.com/COVESA/vss-tools (cit. on p. 19).

[28] digitalplaybook.org. *Overview: playground.digital.auto*. URL: https://www.digitalplaybook.org/index.php?title=Overview:_playground.digital.auto (cit. on p. 40).

[29] digitalplaybook.org. *playground.digital.auto Vehicle API Catalogue*. URL: https://digitalauto.netlify.app/model (cit. on p. 17).

[30] Eclipse Foundation. *Home - iceoryx.io*. URL: https://iceoryx.io/latest/ (cit. on p. 41).

[31] Electronic Design. *What's the Difference Between Domain and Zonal Automotive Architectures?* 2021. URL: https://www.electronicdesign.com/markets/automotive/article/21166567/electronic-design-whats-the-difference-between-domain-and-zonal-automotive-architectures (cit. on p. 13).

[32] EPAM Systems, Renesas Electronics Corporation. *Aos – Open Functions Management System*. URL: https://aoscloud.io/de/ (cit. on p. 40).

[33] franca. *Franca*. URL: https://franca.github.io/franca/ (cit. on p. 19).

[34] Google LLC. *Protocol Buffers Documentation*. 2023. URL: https://protobuf.dev/ (cit. on p. 19).

[35] GraphQL Foundation. *GraphQL | A query language for your API*. 2023. URL: https://graphql.org/ (cit. on p. 19).

[36] A. Gulbrandsen, D. L. Esibov. *A DNS RR for specifying the location of services (DNS SRV)*. RFC 2782. Feb. 2000. DOI: 10.17487/RFC2782. URL: https://www.rfc-editor.org/info/rfc2782 (cit. on p. 32).

[37] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. Oct. 2012. DOI: 10.17487/RFC6749. URL: https://www.rfc-editor.org/info/rfc6749 (cit. on p. 19).

[38] S. Husnjak, D. Peraković, I. Forenbacher, M. Mumdziev. "Telematics system in usage based motor insurance". In: *Procedia Engineering* 100 (2015), pp. 816–825 (cit. on p. 40).

[39] *Internet Protocol*. RFC 791. Sept. 1981. DOI: 10.17487/RFC0791. URL: https://www.rfc-editor.org/info/rfc791 (cit. on p. 70).

[40] International Organization for Standardization. *Road Vehicles — Extended vehicle (ExVe) methodology — Part 1: General information*. Standard. Geneva, CH: International Organization for Standardization, Dec. 2017 (cit. on p. 40).

[41] International Organization for Standardization. *Road Vehicles — Extended vehicle (ExVe) methodology — Part 2: Methodology for designing the extended vehicle*. Standard. Geneva, CH: International Organization for Standardization, Jan. 2018 (cit. on p. 40).

[42] International Organization for Standardization. *Road vehicles — Extended vehicle (ExVe) web services — Part 1: Content and definitions*. Standard. Geneva, CH: International Organization for Standardization, Nov. 2021 (cit. on p. 40).

[43] International Organization for Standardization. *Road vehicles — Extended vehicle (ExVe) web services — Part 2: Access*. Standard. Geneva, CH: International Organization for Standardization, Nov. 2021 (cit. on p. 40).

[44] International Organization for Standardization. *Road vehicles — Information for remote diagnostic support — General requirements, definitions and use cases — Amendment 1*. Standard. Geneva, CH: International Organization for Standardization, Nov. 2021 (cit. on p. 40).

[45] International Organization for Standardization. *Data elements and interchange formats — Information interchange — Representation of dates and times*. Standard. Geneva, CH: International Organization for Standardization, Dec. 2004 (cit. on p. 20).

[46] M. Johanson, P. Dahle, A. Soderberg. "Remote vehicle diagnostics over the internet using the DoIP protocol". In: *Proceedings of the Sixth International Conference on Systems and Networks Communications*. IARIA Barcelona, Spain. 2011, pp. 226–231 (cit. on p. 13).

[47] M. B. Jones, J. Bradley, N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. May 2015. DOI: 10.17487/RFC7519. URL: https://www.rfc-editor.org/info/rfc7519 (cit. on p. 20).

[48] F. J. Koch. "Opportunities and Barriers for Advancing the API Economy within the Automotive Industry". In: *Technische Universität München: Munich, Germany* (2019) (cit. on p. 40).

[49] W. Lee, U. Bjorkengren. *VISS version 2-Transport*. W3C Working Draft. https://www.w3.org/TR/2023/WD-viss2-transport-20230418/. W3C, Apr. 2023 (cit. on pp. 13, 19, 21–26, 46).

[50] W. Lee, U. Bjorkengren. *VISS version 2-Transport*. W3C Working Draft. https://www.w3.org/TR/2023/WD-viss2-transport-20230908/. W3C, Sept. 2023 (cit. on p. 111).

[51] T. Lin, D. F. Blanco, J. Guariguata. *Communication Management in Automotive Service-Oriented Architectures*. IEEE SA Ethernet and IP at Automotive Technology Day. Nov. 2021. URL: https://standards.ieee.org/wp-content/uploads/import/documents/other/eipatd-presentations/2021/d1-02.pdf (cit. on p. 41).

[52] N. Lu, N. Cheng, N. Zhang, X. Shen, J. W. Mark. "Connected vehicles: Solutions and challenges". In: *IEEE internet of things journal* 1.4 (2014), pp. 289–299 (cit. on p. 40).

[53] K. Matheus, T. Königseder. *Automotive ethernet*. Cambridge University Press, 2021 (cit. on pp. 13, 26, 41).

[54] A. Mayr, M. Helmling. "Middleware Protocols in the Automobile: Service-Oriented, Data-Centric or RESTful?" In: *Elektronik automotive* (Mar. 2020). URL: https://cdn.vector.com/cms/content/know-how/_technical-articles/PREEvision/PREEvision_MiddlewareProtocols_ElektronikAutomotive_202003_PressArticle_EN.pdf (cit. on p. 41).

[55] A. Melnikov, I. Fette. *The WebSocket Protocol*. RFC 6455. Dec. 2011. DOI: 10.17487/RFC6455. URL: https://www.rfc-editor.org/info/rfc6455 (cit. on pp. 17, 19).

[56] Mercedes-Benz Connectivity Services GmbH. *Mercedes–Benz /developers – The API platform by Mercedes-Benz*. URL: https://developer.mercedes-benz.com/ (cit. on p. 40).

[57] Microsoft. *JSON · microsoft/cpprestsdk Wiki · GitHub*. 2017. URL: https://github.com/microsoft/cpprestsdk/wiki/JSON (cit. on p. 75).

[58] H. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. DOI: 10.17487/RFC2616. URL: https://www.rfc-editor.org/info/rfc2616 (cit. on pp. 17, 25).

[59] T. Nomura, A. Katsuyuki. *What is the conqueror in the SOA platform for the future in-vehicle networks? - A study based on JASPAR's automotive use cases*. IEEE SA Ethernet and IP at Automotive Technology Day. Nov. 2021. URL: https://standards.ieee.org/wp-content/uploads/import/documents/other/eipatd-presentations/2021/additional-presentation.pdf (cit. on p. 41).

[60] OASIS. *MQTT Version 5.0*. Standard. Mar. 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf (cit. on pp. 17, 19).

[61] OMG. *OMG Data Distribution Service (DDS) Version 1.4*. formal/2015-04-10. Mar. 2015. URL: https://www.omg.org/spec/DDS/1.4/PDF (cit. on p. 17).

[62] R. Palin, D. Ward, I. Habli, R. Rivett. "ISO 26262 safety cases: Compliance and assurance". In: (2011) (cit. on p. 13).

[63] E. Rescorla. *HTTP Over TLS*. RFC 2818. May 2000. DOI: 10.17487/RFC2818. URL: https://www.rfc-editor.org/info/rfc2818 (cit. on p. 19).

[64] M. Rumez, D. Grimm, R. Kriesten, E. Sax. "An overview of automotive service-oriented architectures and implications for security countermeasures". In: *IEEE access* 8 (2020), pp. 221852–221870 (cit. on p. 41).

[65] D. C. Schmidt. "Reactor: an object behavioral pattern for demultiplexing and dispatching handles for synchronous events". In: *Proceedings of the First Pattern Languages of Programs Conference*. 1994 (cit. on p. 76).

[66] M. Steffelbauer. "SOVD-The Diagnostic Standard of Tomorrow". In: *ATZelectronics worldwide* 16.4 (2021), pp. 44–48 (cit. on pp. 13, 40).

[67] The Eclipse Foundation. *Eclipse KUKSA*. URL: `https://eclipse.github.io/kuksa.website/` (cit. on p. 40).

[68] D. Tomaszuk, G. Kellogg. *RDF 1.2 Turtle*. W3C Working Draft. https://www.w3.org/TR/2023/WD-rdf12-turtle-20230831/. W3C, Aug. 2023 (cit. on p. 19).

[69] Vector Informatik GmbH. *MICROSAR Adaptive*. 2023. URL: `https://www.vector.com/de/de/produkte/produkte-a-z/embedded-components/adaptive-microsar/` (cit. on p. 75).

[70] Verband der Automobilindustrie e. V. *ADAXO: Automotive Data Access – Extended and Open*. Position. Berlin, DE, Dec. 2021. URL: `https://www.vda.de/dam/jcr:72f7590f-cd77-4e8c-b8dd-d34df3c483f9/VDA_5690_Positionspapier_ADAXO_EN_RZ.pdf?mode=view` (cit. on p. 40).

[71] W3C. *W3C Automotive Interface Implementation - WAII*. URL: `https://github.com/w3c/automotive-viss2` (cit. on pp. 20, 40).

[72] N. Weiss, M. Schreieck, M. Wiesche, H. Krcmar. "From Product to Platform: How can BMW compete with Platform Giants?" In: *Journal of Information Technology Teaching Cases* 11.2 (2021), pp. 90–100 (cit. on p. 40).

[73] F. Yergeau. *UTF-8, a transformation format of ISO 10646*. RFC 3629. Nov. 2003. DOI: `10.17487/RFC3629`. URL: `https://www.rfc-editor.org/info/rfc3629` (cit. on p. 46).