# Introduction

## What Is SOME/IP Protocol?

**Scalable Service-Oriented Middleware over IP** (SOME/IP) is a **communication protocol, commonly used in embedded and automative**. Running on top of UDP or TCP and the underlying serialization/wire format, it provides a standardized way for services to discover, connect, and exchange data (via RPCs) with each other.

## Signal-Oriented vs. Service-Oriented Data Transmission

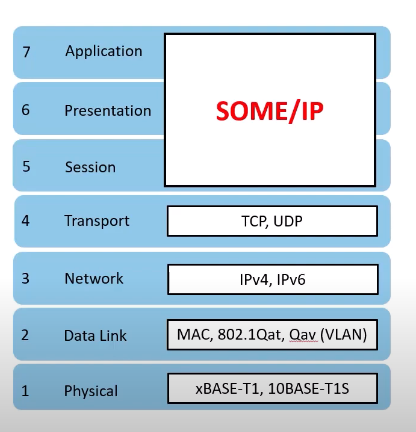
|  |  |  |
| --- | --- | --- |
|  | **Signal-Oriented Data Transmission** | **Service-Oriented Data Transmission** |
| **Data transmission** | A sender **sends data when it sees a need**, such as when values are updated, independent of whether these data are currently needed by a receiver in the network. | A sender **only sends data when a receiver** in the network needs this data. So the network and all connected nodes are not loaded by unnecessary data. |
| **Example** | CAN, LIN, FlexRay, MOST |  |
| **Resource** | Shared medium. Shared badwidth | Switch medium. Higher bandwith (100Mbit/s) |
| **Effectiveness** | 1:1 costs the same as 1:n communication | Effective unicast communication |
| **Message size** | Small | Larger messages (0-1400 bytes) |
| **Scalability** | Scales by adding another CAN or FR bus. | Scales by adding additional links |

## OSI Layer vs. SOME/IP

OSI (Open Systems Interconnection) model is a conceptual framework that defines the functions and interactions of different layers in a network architecture. It consists of seven layers, each responsible for specific tasks such as *physical transmission*, *logical addressing*, and *application-level protocols*.

In the case of SOME/IP, it is built upon the *transport layer protocols*, such as TCP and UDP. And it primarily operates at the *application layer* of the OSI model.

Following is OSI layer and its mapping to SOME/IP:

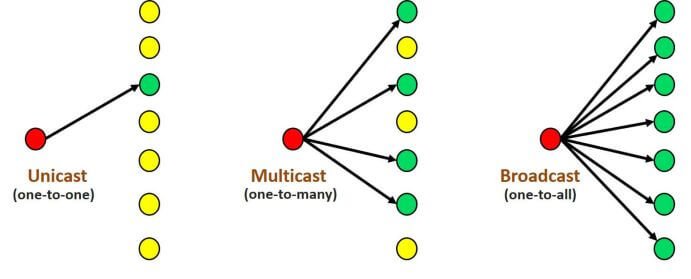


## TCP vs. UDP in SOME/IP

SOME/IP runs on top of TCP/IP or UDP/IP. But UDP is prefered because:

* UDP does not have timeout or retry mechanisms that can be troublesome for real-time control in automotive applications.
* With UDP, when an event occurs the server can **send data to all clients** **with an active subscription** via Unicast, Multicast, or Broadcast. In other words, multiple clients can subscribe to content of a service simultaneously. If the content is made available via TCP, a connection to the server must be established by each client, which enables the respective sending of the data.

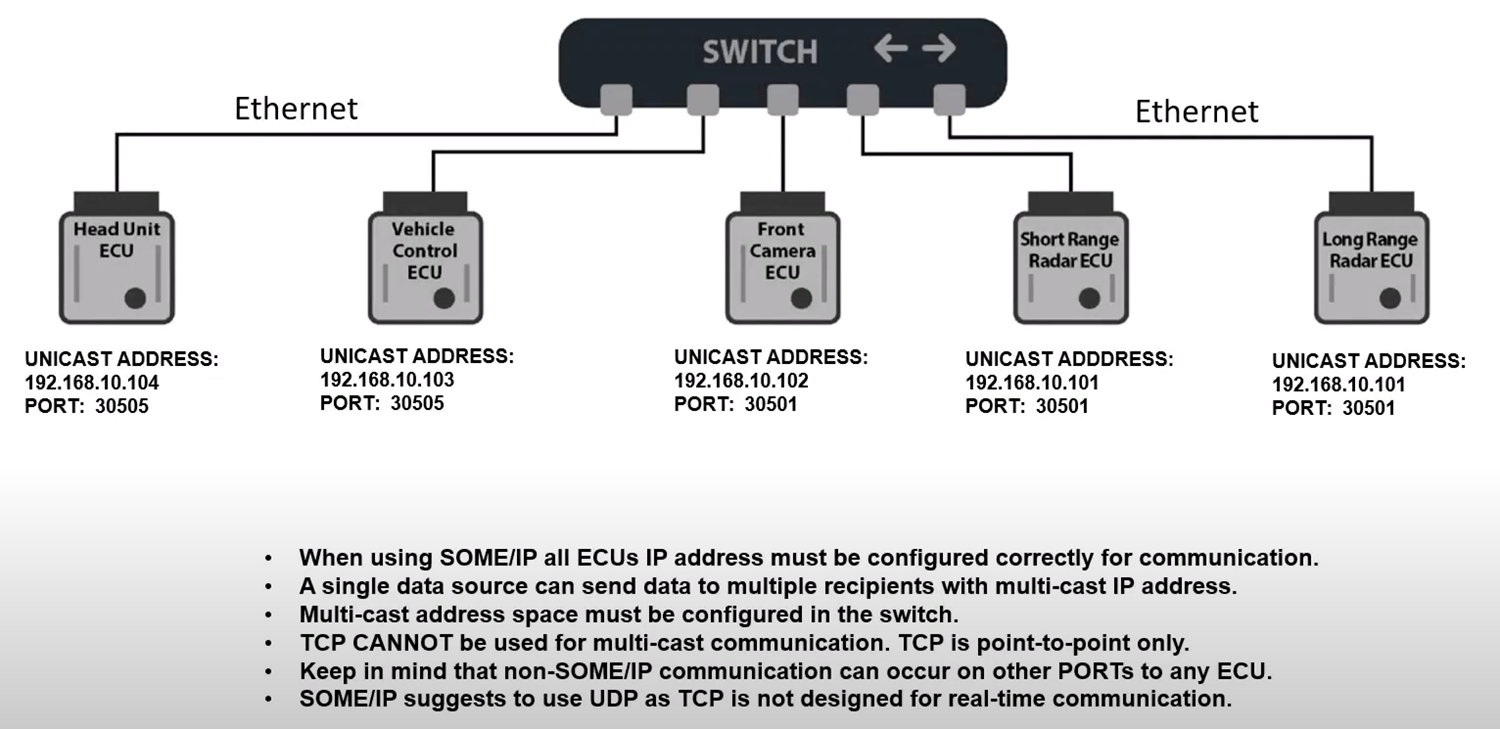
However, UDP limits the message size to **1400 bytes**. So, in case very long payload is necessary, TCP is prefered.



## AUTOSAR vs. SOME/IP

**SOME/IP is compatible with AUTOSAR** at least on the wire-format level; i.e. can communicate with PDUs AUTOSAR, can receive and send without modification to the AUTOSAR standard, can be used for inter-ECU Client/Server Serialization, etc. The mappings within AUTOSAR shall be chosen according to the SOME/IP specification.

# SOME/IP Configuation



# Glossary

|  |  |
| --- | --- |
| **Abbreviation/ Acronym** | **Description** |
| SOME/IP | Scalable Service-Oriented Middleware over IP.  The only valid abbreviation is SOME/IP. Other abbreviations (e.g. ~~Some/IP~~) are wrong and shall not be used. |
| Service | Any combination of zero or more Fields, Events, Methods. |
| Service Interface | The formal specification of the service including its methods, events, and fields. |
| Service Instance | Implementation of a service, which can exist more than once in the vehicle and more than once on an ECU. |
| Server | The ECU **offering** a service instance. |
| Client | The ECU **using** the service instance. |
| Producer | Also called Provider. This **has** the service implementation. |
| Consumer | Also called Subscriber. This **uses** the service implementation. |
| Field | A field does represent a status and thus has an valid value at all times on which getter, setter and notifier act upon. |
| Getter | A Request/Response call that allows **read access to a field**. |
| Setter | A Request/Response call that allows **write access to a field**. |
| Notifier | Sends out event message with a new value on change of the value of the field. |
| Event | A uni-directional data transmission that is **only invoked on changes or cyclically**, and is sent from the producer to the consumers. |
| Notification Event | An event message of the notifier of a field. |
| Eventgroup | A logical grouping of events and notification events of fields inside a service in order to allow subscription. |
| Method | A method, procedure, function, or subroutine that is called/invoked. |
| Parameters | An input, output, or input/output arguments of a method or an event. |
| Remote Procedure Call (RPC) | A **method call from one ECU to another** that is transmitted using messages. |
| Request | A message of the client to the server invoking a method. |
| Response | A message of the server to the client transporting result of a method invocation. |
| Fire and Forget | Requests without response message. |
| Union | A data structure that dynamically assumes different data types. |
| Extensible struct | A struct which is serialized with tags. New members can be added in a compatible way at arbitrary positions and optional members are possible. |
| Non-extensible (standard) struct | A struct which is serialized without tags. At most, new members can be added in a compatible way at the end of the struct and optional members are not possible. |
| Byte Order Mark | The byte order mark (BOM) is a Unicode character, U+FEFF, whose appearance as a magic number at the start of a text stream is used to indicate the used encoding. |
| SOME/IP-TP | SOME/IP Transport Protocol |
| SOME/IP-SD | SOME/IP Service Discovery |

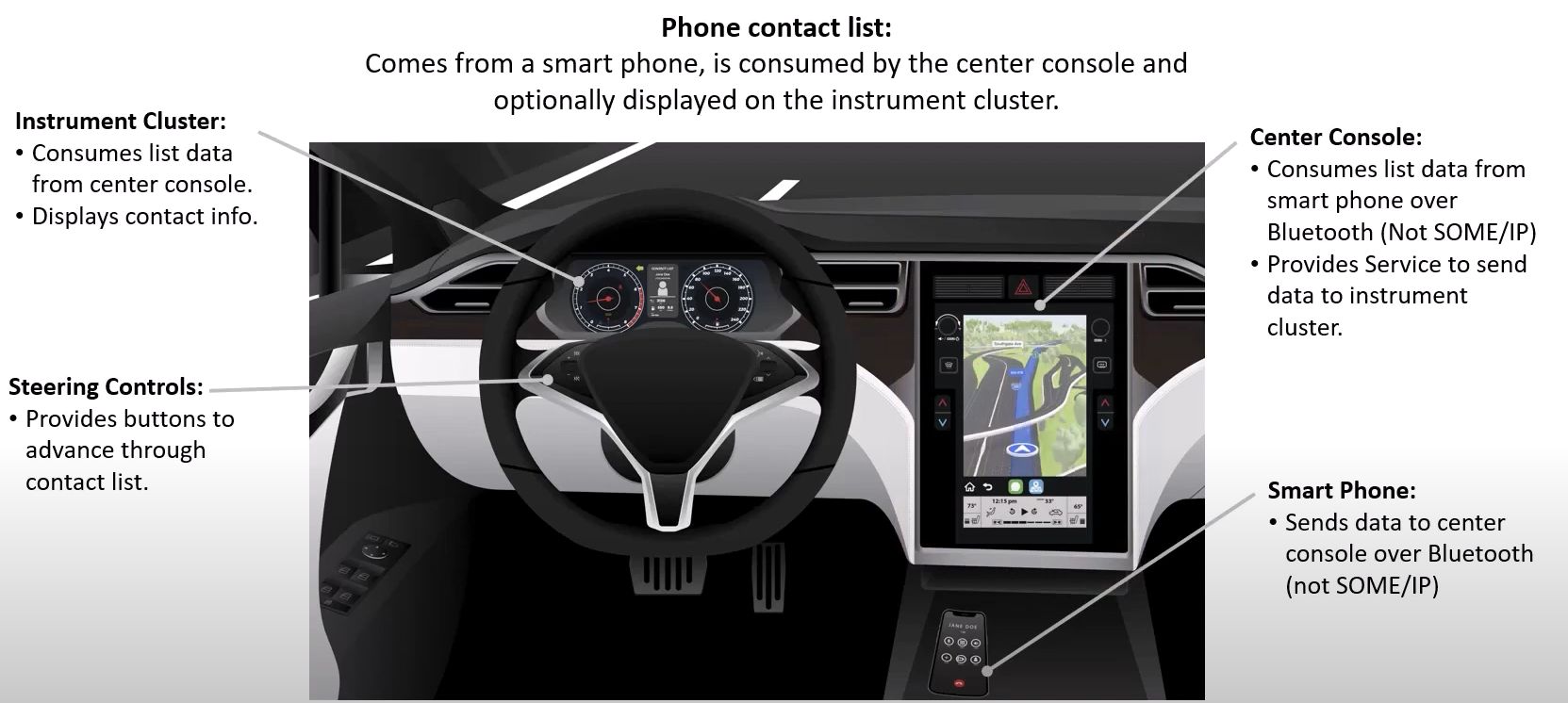
# SOME/IP Example

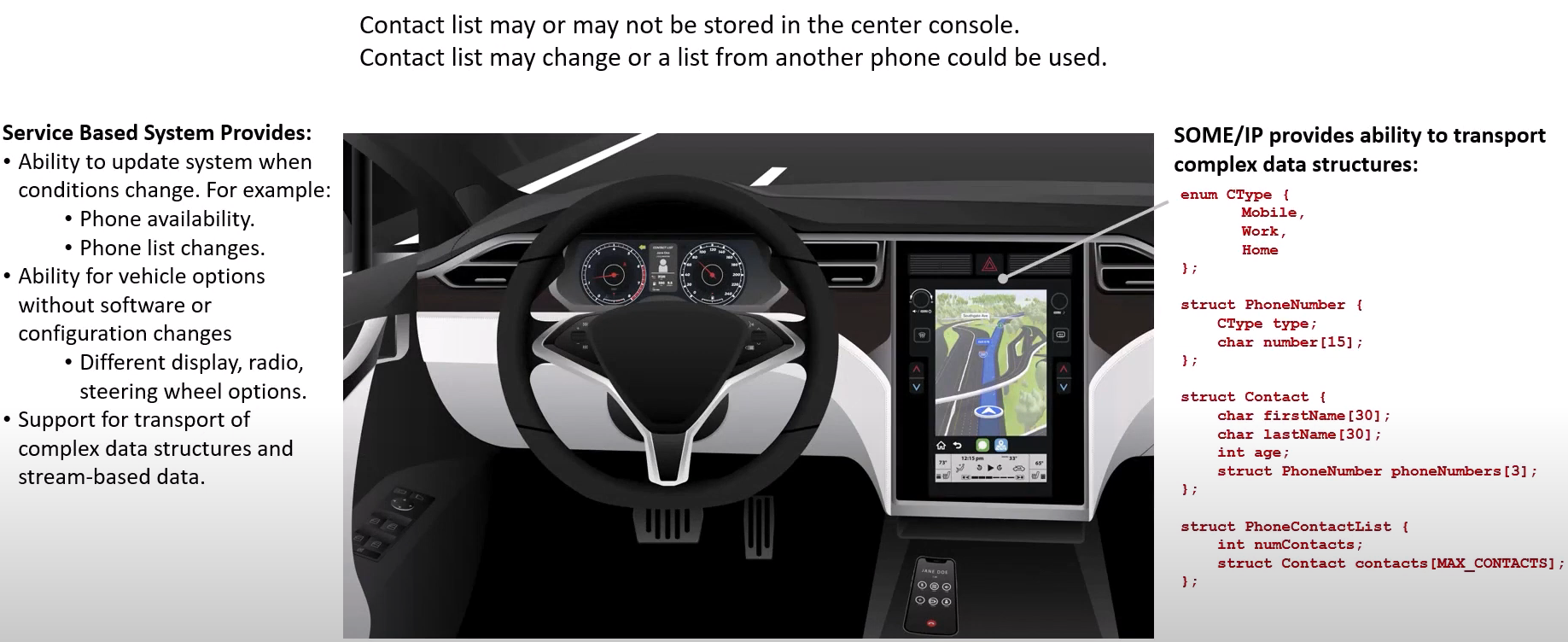
A smartphone contact list is a good example application.

Here we have a smartphone introduced into the vehicle and the center console often has the ability to consume the phone contact list into its memory over Bluetooth. While driving the car, the user may want to see this contact list in the instrument cluster and use the steering wheel controls to advance through the contact and make a phone call.

In this example, the center console provides a service of a phone contact list to the instrument cluster. This example illustrates some of the advantages of a service based architecture:

* Services can exist, and come and go based on the conditions of the vehicle. In our example, maybe there's no cell phone in the car and therefore the service of a phone contact list doesn't make sense to even exist.
* Phone contact list information can be some complex data. First name, last name, several phone numbers, addresses.A service-based architecture allows for this complex information, in structures or arrays, to be transferred across the vehicle network.





# Communication Patterns

There are only two primary communication patterns in SOME/IP protocol: **Request/Response** and **Notification**. But for these, we can have several variants.

## Request/Response

(Method Call)

|  |  |  |
| --- | --- | --- |
| A client sends a request to a server and receives a corresponding response. The request is a message with a specific method ID and payload.  This pattern is widely used in scenarios where a client requires a specific service or data from a server. It ensures **synchronous** communication between services because the client can pause its execution until it receives the response. |  | @startuml  hide footbox  skinparam monochrome true  skinparam ParticipantPadding 20  participant "Client\n(Consumer)" as client  participant "Server\n(Provider)" as server  client-> server: Request (param)  |||  server -> client: Response (result)  |||  @enduml |

## Fire-and-Forget Request

(Method Call Without Return)

|  |  |  |
| --- | --- | --- |
| The client does not expect any response from the server. The request is a message with a specific method ID and payload.  This pattern is commonly used in scenarios where a client needs to send a command or trigger an action but does not require any specific response or confirmation. The client can continue its execution without waiting for a reply. By **reducing the overhead of waiting for a response**, this pattern improves performance and efficiency. |  | @startuml  hide footbox  skinparam monochrome true  skinparam ParticipantPadding 20  participant "Client\n(Consumer)" as client  participant "Server\n(Provider)" as server  client-> server: Request (param)  |||  @enduml |

## Field Getter/Setter

(Method Call)

|  |  |  |
| --- | --- | --- |
| Similar to the Request/Response pattern, the client sends a request to a server and receives a corresponding response. Only difference is that the request is associated with a Field only. |  | @startuml  hide footbox  skinparam monochrome true  skinparam ParticipantPadding 20  participant "Client\n(Consumer)" as client  participant "Server\n(Provider)" as server  client-> server: Get/Set (param)  |||  server -> client: Response (result)  |||  @enduml |

## Event Notification

(Callback)

|  |  |  |
| --- | --- | --- |
| ~~The client subscribes to specific event~~ **~~groups~~** ~~and receive notifications~~ **~~asynchronously~~** ~~whenever those events occur (~~**~~on change~~** ~~or~~ **~~cyclically~~** ~~or both as per the sender configuration).~~  ~~The service can act as a publisher or a subscriber. Publishers are responsible for generating events and notifying the subscribers, while subscribers express their interest in specific events and receive the corresponding notifications. In this case, the publisher is server and the subscribe is client.~~  ~~The pattern is crucial in scenarios where timely updates and notifications are required. For example, in an automotive system, a service might subscribe to the "Engine Temperature" event and receive notifications whenever the temperature exceeds a certain threshold. This enables~~ **~~real-time monitoring~~** ~~and immediate action in critical situations.~~ |  | @startuml  hide footbox  skinparam monochrome true  skinparam ParticipantPadding 20  participant "Client\n(Consumer)" as client  participant "Server\n(Provider)" as server  client-> server: Service Discovery\n(Subscribe Event)  server-> client: Service Discovery ACK\n(Subscribe Event ACK)  |||  server-> client: Event Notification (result)  |||  server-> client: Event Notification (result)  |||  server-> client: ...  @enduml |

## Event Group Notification

(Callback)

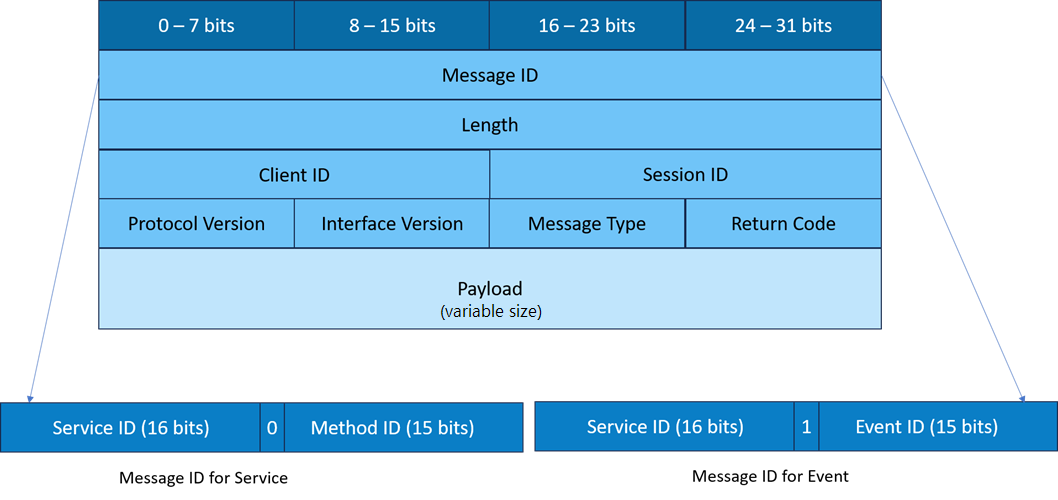
|  |  |  |
| --- | --- | --- |
| Sometimes, it makes sense to **group a set of events** logically so that the subscribers can subscribe to this event group. With this, they can get information **on change from any** of these events with a single subscription rather than requesting for each of the constituent events. For example, all the events from the Fuel Computer modules – Low Fuel, Tank Fill, Fuel Drain, etc. can be grouped under the FuelEvents and used by the clients. SOME/IP supports such event groups to optimize communication. |  | @startuml  hide footbox  skinparam monochrome true  skinparam ParticipantPadding 20  participant "Client\n(Consumer)" as client  participant "Server\n(Provider)" as server  client-> server: Service Discovery\n(Subscribe Eventgroup)  server-> client: Service Discovery ACK\n(Subscribe Eventgroup ACK)  |||  server-> client: Eventgroup Notification (result)  |||  server-> client: Eventgroup Notification (result)  |||  server-> client: ...  @enduml |

## Field Notification

(Callback)

|  |  |  |
| --- | --- | --- |
| Like the Event Notification and the Event Group Notification pattern, this one is **asynchronous** and occurs **on change.** Only difference is that this is associated with a Field only, and hence it's possible to invoke a *getter* or *setter* on to it.  For example, in an automotive system, a service might have fields like "Engine Speed," "Fuel Level," and "Odometer Reading." These fields can be updated and accessed independently, and when there is a change, the client are alerted along with the updated value.  This optimizes the communication process and **reduces unnecessary data transfer**, leading to improved performance and efficiency. |  | @startuml  hide footbox  skinparam monochrome true  skinparam ParticipantPadding 20  participant "Client\n(Consumer)" as client  participant "Server\n(Provider)" as server  client-> server: Service Discovery\n(Subscribe Eventgroup)  server-> client: Service Discovery ACK\n(Subscribe Eventgroup ACK)  |||  server-> client: Field Notification (result)  |||  server-> client: Field Notification (result)  |||  server-> client: ...  |||  @enduml |

# SOME/IP Message Structure



Each SOME/IP message consists of a **header** and a **payload**. The header contains essential information, such as the message ID, service ID, and method ID, which helps in routing and processing the message correctly. The payload, on the other hand, carries the actual data that needs to be exchanged between services.

### Header

The first 32 bits (4 bytes) constitutes the *Message ID*. This Message ID must be unique across the vehicle network and can be organized in two ways based on the 15th bit (MS bit of the lower half word).

* If this bit is 0, then the Message ID is used to represent the Remote Procedural Call (RPC) with the upper 16 bit constituting the *Service ID* and the lower 15 bits indicating the method ID within this service.
* If this bit is 1, then the Message ID is used to represent the Event with the upper 16 bits, again used for service, and the lower 15 bits for the *Event ID*.

The next 32 bits indicate the *Message Length* including the header minus 8 (to exclude message ID and length fields).

The next part represents the *Request ID* which is typically a concatenation of *Client ID* and *Session ID*. The Client ID represents unique elements in the system and Session ID used for differentiating requests originating from the same client.

The following 8 bits is used to hold the *Protocol Version* (currently, 1), followed by a 8-bit *Interface Version*. The next 8 bits hold the *Message Type*, and the last 8 bits hold the *Return Code*. You can check the following tables for predefined values for Message Type and Return Code:

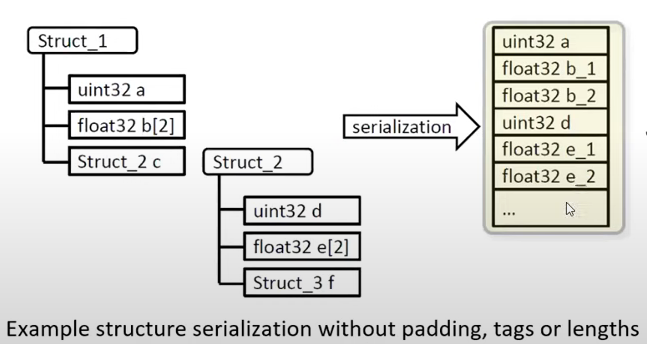
|  |  |
| --- | --- |
|  |  |

### Payload

The payload can vary depending on the specific method being used. For example, in the Request/Response method, the payload contains the request data sent by the client and the response data returned by the server. In the Fire-and-Forget method, the payload only contains the request data, as no response is expected.

# SOME/IP Message Serialization (or Payload Transformer)

One of the beauties of SOME/IP is the ability to handle complex data structure using data serialization. Serialization is the process of converting structured data **into byte streams to transit on-wire** over a network. And then unpack it at the other end.



# SOME/IP Service Discovery

## What Is It?

A client has the option of **subscribing to the content of a service** on the server. If an event occurs, it receives the updated data from the server. The SOME/IP Service Discovery (SOME/IP-SD) is the term to describe such the subscription.

Especially, services can subscribe each other **dynamically** (at runtime). This eliminates the need for hard-coded IP addresses and simplifies the integration of new services into an existing system.

After the connection between server and client has been completed, the data exchange with SOME/IP messages can begin.

## SOME/IP-SD Message Flow

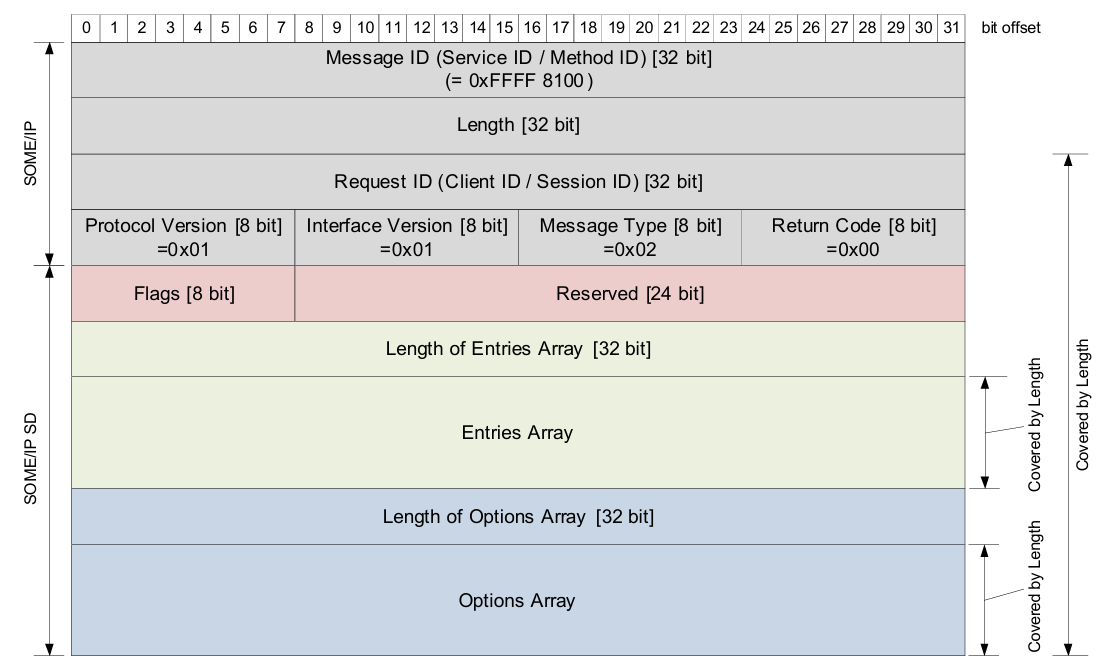
SOME/IP-SD provides two mechanisms that allow the **dynamic** discovery of services:

* *Offer Service* enables the server to provide the network with all of its offered services.
  + The server broadcasts (multicasts) messages containing all the services it offers

|  |  |
| --- | --- |
|  | @startuml  hide footbox  skinparam ParticipantPadding 20  participant "Client\n(Consumer)" as client  participant "Server\n(Producer)" as server  server-> client: <color green>Offer Service  |||  client-> server: <color green>Subscribe Eventgroup  server-> client: <color green>Subscribe Eventgroup ACK  |||  server-> client: Eventgroup Notification (result)  |||  server-> client: Eventgroup Notification (result)  |||  server-> client: ...  |||  client-> server: <color green>Stop Subscribe Eventgroup  |||  server-> client: <color green>Stop Offer Service  @enduml |

* *Find Service* allows the client to request an available service on the server.
  + If services are required by client but at the moment not offered, then "find messages" can be sent.

## SOME/IP-SD Message Structure



More detail: <https://www.youtube.com/watch?v=Wec0lhLMu94&ab_channel=IntrepidControlSystems> (49:40)

# vsomeip

<https://github.com/COVESA/vsomeip>

<https://github.com/COVESA/vsomeip/wiki/vsomeip-in-10-minutes>

**Note**: vsomeip does not implement the serialization of data structures! This is covered by the SOME/IP binding of CommonAPI. vsomeip just covers the SOME/IP protocol and the Service Discovery.

Google Protobuf can be used for as a format for data serialization.

# CommonAPI C++ for SOME/IP

<https://covesa.github.io/capicxx-someip-tools/>

# Open-Source Tool Supports

## Wireshark 3.2

Wireshark supports SOME/IP, SOME/IP-SD, SOME/IP-TP, and configurable SOME/IP payload dissection.

## FibexConverter

A generator for payload dissection configs.

# References

<https://some-ip.com/>

<https://www.youtube.com/watch?v=Wec0lhLMu94&ab_channel=IntrepidControlSystems>

<https://www.autosar.org/fileadmin/standards/R21-11/FO/AUTOSAR_RS_SOMEIPProtocol.pdf>

<https://some-ip.com/papers/cache/AUTOSAR_TR_SomeIpExample_4.2.1.pdf>