Document Title	Example for a Serialization Protocol (SOME/IP)
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	637
Document Classification	Auxiliary

Document Status	Final
Part of AUTOSAR Release	4.2.1

Document Change History

Release	Changed by	Description
4.2.1	AUTOSAR Release Management	• Added SD Peer Identification • Extended Error Handling • Minor corrections and clarifications
4.1.3	AUTOSAR Release Management	• Added support for bitfields • Client ID is configurable • Defined applicability of SOME/IP-SD Options for entry types
4.1.1	AUTOSAR Administration	Initial Release

Disclaimer

Disclaimer

Advice for users

AUTOSAR specifications may contain exemplary items (exemplary reference models, "use cases", and/or references to exemplary technical solutions, devices, processes or software).

Any such exemplary items are contained in the specifications for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such specifications, nor any later documentation of AUTOSAR conformance of products actually implementing such exemplary items, imply that intellectual property rights covering such exemplary items are licensed under the same rules as applicable to the AUTOSAR Standard.

Table of Contents

1 Introduction and functional overview

This document specifies the **Scalable service-Oriented middlewarE over IP (SOME/IP)** — an automotive/embedded RPC mechanism and the underlying serialization / wire format — as an example for a Serializatier called by the RTE.

The only valid abbreviation is SOME/IP. Other abbreviations (e.g. Some/IP) are wrong and shall not be used.

The basic motivation to specify "yet another RPC-Mechanism" instead of using an existing infrastructure/technology is the goal to have a technology that:

- Fulfills the hard requirements regarding resource consumption in an embedded world
- Is compatible through as many use-cases and communication partners as possible
- 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. The mappings within AUTOSAR shall be chosen according to the SOME/IP specification.
- Provides the features required by automotive use-cases
- Is scalable from tiny to large platforms
- Can be implemented on different operating system (i.e. AUTOSAR, GENIVI, and OSEK) and even embedded devices without operating system

SOME/IP is only an example for a serializer which can be used for inter-ECU Client/Server Serialization. An implementation of SOME/IP allows AUTOSAR to parse the RPC PDUs and transport the signals to the application.

As a consequence this example defines several feature sets. The feature set "basic" is compatible to AUTOSAR 4.1.1. The other feature sets are in progress to be integrated into AUTOSAR. The goal is to increase the compatibility towards higher sophisticated feature sets. It is however possible to use these features in non-AUTOSAR nodes or to implement them inside the AUTOSAR application with a carefully designed interface (see Chapter 6.4) and suitable tool chain.

For ECUs not using AUTOSAR the complete feature set can be supported as of today but a limited set of features can be used in the communication with AUTOSAR ECUs.

SOME/IP and SOME/IP-SD may be implemented in AUTOSAR in different modules. Currently the Socket Adaptor may write the Message-ID and Length field by means of the header mode.

For the data path (SOME/IP), the message may be serialized/deserialized by the COM, an pluggable serializer in the RTE, or a proxy SWS.

For the control path (SOME/IP-SD), the Service Discovery Module implements SOME/IP-SD including the SOME/IP headers without the Message-ID and Length field itself.

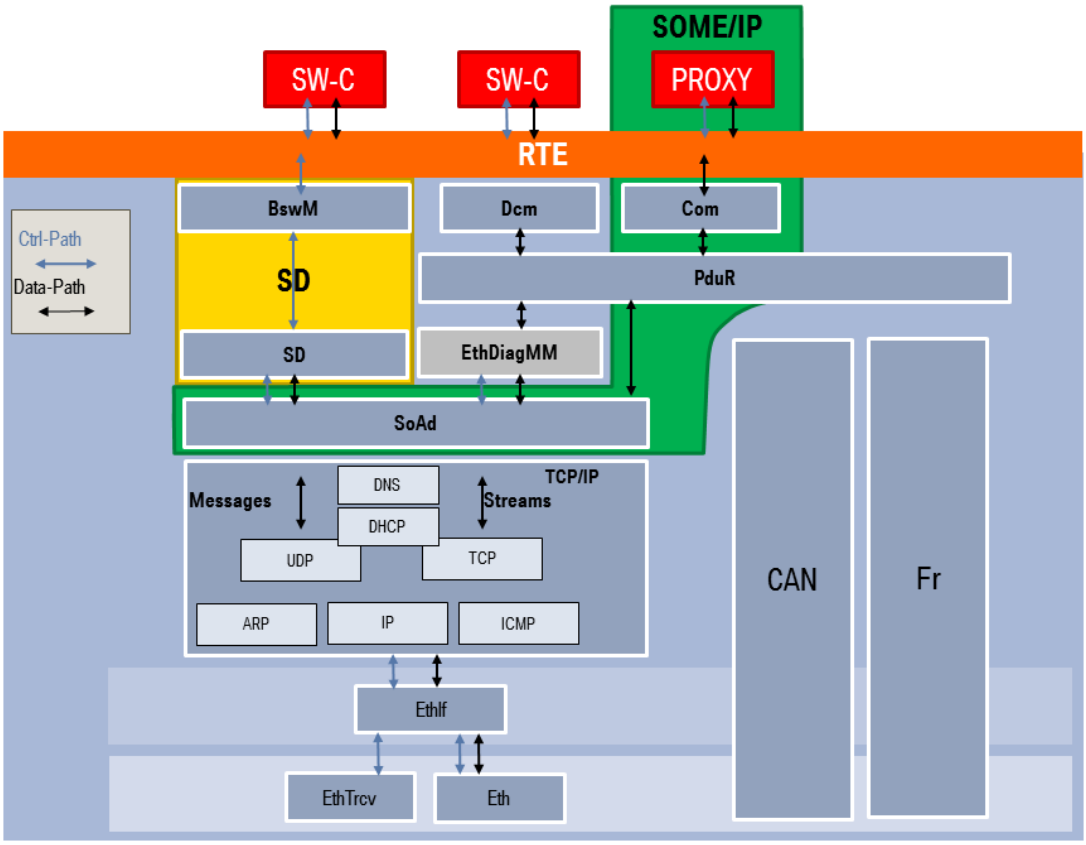

Figure 1.1: Example for SOME/IP in AUTOSAR

2 Acronyms and Abbreviations

The glossary below includes acronyms and abbreviations relevant to the SOME/IP specification that are not included in the [1, AUTOSAR glossary].

Abbreviation / Acronym:	Description:
Method	a method, procedure, function, or subroutine that is called/invoked.
Parameters	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 results of a method invocation
Request/Response communication	a RPC that consists of request and response
Fire&Forget communication	a RPC call that consists only of a request message
Event	a "Fire & Forget callback" that is only invoked on changes or cyclically and is sent from Server to Client.
Field	a field does represent a status and thus has an valid value at all times on which getter, setter and notfier act upon.
Notification Event	an event message the notifier of an field sends. The message of such a notifier cannot be distinguished from the event message; therefore, when refering to the message of an event, this shall also be true for the messages of notifiers of fields.
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.
Service	a logical combination of zero or more methods, zero or more events, and zero or more fields (empty service is allowed, e.g. for announcing non-SOME/IP services in SOME/IP-SD)
Eventgroup	a logical grouping of events and notification events of fields inside a service in order to allow subscription
Service Interface	the formal specification of the service including its methods, events, and fields

Abbreviation / Acronym:	Description:
Service Instance	software implementation of the service interface, which can exist more than once in the vehicle and more than once on an ECU
Server	The ECU offering a service instance shall be called server in the context of this service instance.
Client	The ECU using the service instance of a server shall be called client in the context of this service instance.
Union or Variant	a data structure that dynamically assumes different data types.
Offering a service instance	that one ECU implements an instance of a service and tells other ECUs using SOME/IP-SD that they may use it.
Finding a service instance	to send a SOME/IP-SD message in order to find a needed service instance.
Requiring a service instance	to send a SOME/IP-SD message to the ECU implementing the required service instance with the meaning that this service instance is needed by the other ECU. This may be also sent if the service instance is not running; thus, was not offered yet.
Releasing a service instance	to sent a SOME/IP-SD message to the ECU hosting this service instances with the meaning that the service instance is not longer needed.
Server-Service-Instance-Entry	The configuration and required data of a service instance the local ECU offers, is called Server-Service-Instance-Entry at the ECU offering this service (Server).
Client-Service-Instance-Entry	The configuration and required data of a service instance another ECU offers, is called Client-Service-Instance-Entry at the ECU using this service (Client)
Publishing an eventgroup	to offer an eventgroup of a service instance to other ECUs using a SOME/IP-SD message
Subscribing an eventgroup	to require an eventgroup of a service instance using a SOME/IP-SD message

3 Related documentation

3.1 Input documents

References

[1] Glossary
AUTOSAR_TR_Glossary

[2] Specification of SOME/IP Transformer
AUTOSAR_SWS_SOMEIPTransformer

[3] Specification of Socket Adaptor
AUTOSAR_SWS_SocketAdaptor

[4] Specification of Service Discovery
AUTOSAR_SWS_ServiceDiscovery

3.2 Related standards and norms

Not applicable.

3.3 Related specification

Not applicable.

4 Constraints and assumptions

4.1 Limitations

This document gives a holistic overview over SOME/IP but doesn't state any requirements towards any implementation of BSW modules. It is only informational!

Please be aware that not all parts of SOME/IP are implemented in AUTOSAR.

Some parts of SOME/IP are implemented in [2, SWS SOME/IP Transformer], [3, SWS Socket Adaptor] and [4, SWS Service Discovery].

Other functionality is currently not supported by AUTOSAR. Among others, the following functionality is not implemented in AUTOSAR:

- Exceptions and exception-specific error data structures
- Structs with variable length
- Tunneling of SOME/IP messages through CAN and FlexRay leads to SOME/IP messages without parts of the header inserted by [3, SWS Socket Adaptor]

4.2 Applicability to car domains

No restrictions.

5 Dependencies to other modules

There are not dependencides to AUTOSAR SWS modules.

5.1 File structure

5.1.1 Code file structure

Not applicable.

5.1.2 Header file structure

Not applicable.

6 Functional specification

6.1 Definition of Identifiers

[TR_SOMEIP_00001] ⌈ A service shall be identified using the Service-ID. ⌋*()*

[TR_SOMEIP_00002] ⌈ Service-IDs shall be of type 16 bit length unsigned integer (uint16). ⌋*()*

[TR_SOMEIP_00003] ⌈ The Service-ID of 0xFFFE shall be used to encode non-SOME/IP services. ⌋*()*

[TR_SOMEIP_00005] ⌈ Different services within the same vehicle shall have different Service-IDs. ⌋*()*

[TR_SOMEIP_00006] ⌈ A service instance shall be identified using the Service-Instance-ID. ⌋*()*

[TR_SOMEIP_00007] ⌈ Service-Instance-IDs shall be of type 16 bit length unsigned integer (uint16). ⌋*()*

[TR_SOMEIP_00008] ⌈ The Service-Instance-IDs of 0x0000 and 0xFFFF shall not be used for a service, since 0x0000 is reserved and 0xFFFF is used to describe all service instances. ⌋*()*

[TR_SOMEIP_00009] ⌈ Different service instances within the same vehicle shall have different Service-Instance-IDs.⌋*()*

Note:
This means that two different camera services shall have two different Service-Instance-IDs SI-ID-1 and SI-ID-2. For all vehicles of a vehicle project SI-ID-1 shall be the same. The same is true for SI-ID-2. If considering another vehicle project, different IDs may be used but it makes sense to use the same IDs among different vehicle projects for ease in testing and integration.

[TR_SOMEIP_00010] ⌈ Methods and events shall be identified inside a service using a 16bit Method-ID, which is called Event-ID for events and notifications. ⌋*()*

[TR_SOMEIP_00011] ⌈ Methods shall use Method-IDs with the highest bit set to 0, while the Method-IDs highest bit shall be set to 1 for events and notifications of fields. ⌋*()*

[TR_SOMEIP_00012] ⌈ An eventgroup shall be identified using the Eventgroup-ID. ⌋*()*

[TR_SOMEIP_00013] ⌈ Eventgroup-IDs shall be of 16 bit length unsigned integer (uint16). ⌋*()*

[TR_SOMEIP_00014] ⌈ Different eventgroups of a service shall have different Eventgroup-IDs. ⌋*()*

6.2 Specification of the SOME/IP on-wire format

Serialization describes the way data is represented in protocol data units (PDUs) transported over an IP-based automotive in-vehicle network.

6.2.1 Transport Protocol

[TR_SOMEIP_00016] ⌈ SOME/IP shall be transported using UDP and TCP based on the configuration. When used in a vehicle the ports used shall be specified in the Interface Specification. ⌋*()*

[TR_SOMEIP_00021] ⌈ If an ECU needs to dynamically use a port number, it shall follow the rules of IETF and IANA for that:

- Ephemeral ports from range 49152-65535

⌋*()*

If not specified otherwise by the Interface Specification (i.e. FIBEX or ARXML), the SOME/IP implementation may use port 30491 as SOME/IP dynamic client port and the port 30501 as first SOME/IP server port. For further server instances the ports 30502, 30503, and so on may be used.

[TR_SOMEIP_00017] ⌈ The IP addresses and port numbers an ECU shall use, shall be taken from the Interface Specification. ⌋*()*

[TR_SOMEIP_00018] ⌈ The client shall take the IP address and port number the server announces using SOME/IP-SD (see Chapter 6.7.3.4.3). ⌋*()*

[TR_SOMEIP_00019] ⌈ SOME/IP-SD currently uses port 30490[1] but this shall be overwritten if another port number is specified in the Interface Specification. ⌋*()*

[TR_SOMEIP_00020] ⌈ The port 30490[2] (UDP and TCP as well) shall be only used for SOME/IP-SD and not used for applications communicating over SOME/IP. ⌋*()*

[TR_SOMEIP_00022] ⌈ It is recommended to use UDP for as many messages as possible and see TCP as fall-back for message requiring larger size. UDP allows the application to better control of timings and behavior when errors occur. ⌋*()*

6.2.1.1 Message Length Limitations

[TR_SOMEIP_00023] ⌈ In combination with regular Ethernet, IPv4 and UDP can transport packets with up to 1472 Bytes of data without fragmentation, while IPv6 uses additional 20 Bytes. Especially for small systems fragmentation shall be avoided, so the SOME/IP header and payload shall be of limited length. The possible usage of security protocols further limits the maximal size of SOME/IP messages. ⌋*()*

[1] Port number 30490 is currently not registered with IANA and might change without further notice.
[2] Port number 30490 is currently not registered with IANA and might change without further notice.

[TR_SOMEIP_00024] ⌈ When using UDP as transport protocol SOME/IP messages shall use up to 1416 Bytes for the SOME/IP header and payload, so that 1400 Bytes are available for the payload. ⌋*()*

The usage of TCP allows for larger streams of data to transport SOME/IP header and payload. However, current transport protocols for CAN and FlexRay as well as AUTOSAR limit messages to 4095 Bytes. When compatibility to those has to be achieved, SOME/IP messages including the SOME/IP header shall not exceed 4095 Bytes.

[TR_SOMEIP_00026] ⌈ See also [TR_SOMEIP_00061] and [TR_SOMEIP_00139] for payload length. ⌋*()*

6.2.1.1.1 AUTOSAR restrictions

[TR_SOMEIP_00027] ⌈ See AUTOSAR SWS COM Chapter 7.6. ⌋*()*

6.2.2 Endianess

[TR_SOMEIP_00028] ⌈ All RPC-Headers shall be encoded in network byte order (big endian) [RFC 791]. The byte order of the parameters inside the payload shall be defined by the interface definition (i.e. FIBEX or ARXML) and shall be in network byte order when possible and if no other byte order is specified. ⌋*()*

[TR_SOMEIP_00029] ⌈ This means that Length and Type fields shall be always in network byte order. ⌋*()*

6.2.3 Header

[TR_SOMEIP_00030] ⌈ For interoperability reasons the header layout shall be identical for all implementations of SOME/IP and is shown in the Figure 6.1. The fields are presented in transmission order; i.e. the fields on the top left are transmitted first. In the following sections the different header fields and their usage is being described. ⌋*()*

[TR_SOMEIP_00031] ⌈

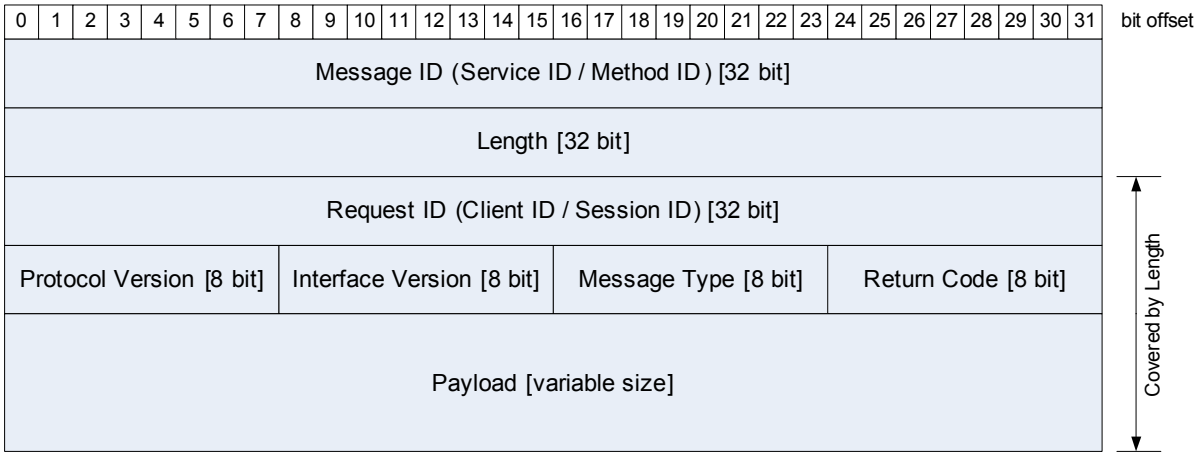

Figure 6.1: SOME/IP Header Format

⌋()

6.2.3.1 IP-Address / port numbers

[TR_SOMEIP_00032] ⌈ The Layout in Figure 6.1 shows the basic header layout over IP and the transport protocol used. This format can be easily implemented with AUTOSAR as well. For details regarding the socket handling see AUTOSAR Socket Adaptor SWS. ⌋()

6.2.3.1.1 Mapping of IP Addresses and Ports in Response and Error Messages

[TR_SOMEIP_00033] ⌈ For the response and error message the IP addresses and port number of the transport protocol shall match the request message. This means:

- Source IP address of response = destination IP address of request.
- Destination IP address of response = source IP address of request.
- Source port of response = destination port of request.
- Destination port of response = source port of request.
- The transport protocol (TCP or UDP) stays the same.

⌋()

6.2.3.2 Message ID [32 Bit]

[TR_SOMEIP_00034] ⌈ The Message ID is a 32 Bit identifier that is used to dispatch the RPC call to a method of an application and to identify an event. The Message ID has to uniquely identify a method or event of a service. ⌋()

[TR_SOMEIP_00035] ⌈ The assignment of the Message ID is up to the user; however, the Message ID has to be unique for the whole system (i.e. the vehicle). The Message ID can be best compared to a CAN ID and should be handled with a comparable process. The next section describes how structure the Message IDs in order to ease the organization of Message IDs. ⌋*()*

6.2.3.2.1 Structure of the Message ID

[TR_SOMEIP_00036] ⌈ An event shall be part of zero to many eventgroups and an eventgroup shall contain zero to many events. A field shall be part of zero to many eventgroups and an eventgroup can contain zero to many fields. ⌋*()*

[TR_SOMEIP_00037] ⌈ Currently empty eventgroups are not used and events as well as fields are mapped to at least one eventgroup. ⌋*()*

[TR_SOMEIP_00038] ⌈ For RPC calls we structure the ID in 2^{16} services with 2^{15} methods:

Service ID [16 Bit]	0 [1 Bit]	Method ID [last 15 Bit]

⌋*()*

[TR_SOMEIP_00039] ⌈ With 16 Bit Service-ID and a 16 Bit Method-ID starting with a 0-Bit, this allows for up to 65536 services with up to 32768 methods each. ⌋*()*

[TR_SOMEIP_00040] ⌈ Since events and notifications (see Notification or Publish/Subscribe) are transported using RPC, the ID space for the events is further structured:

Service ID [16 Bit]	1 [1 Bit]	Event ID [last 15 Bit]

⌋*()*

[TR_SOMEIP_00041] ⌈ This means that up to 32768 events or notifications per service are possible. ⌋*()*

6.2.3.3 Length [32 Bit]

[TR_SOMEIP_00042] ⌈ Length is a field of 32 Bits containing the length in Byte of the payload beginning with the Request ID/Client ID until the end of the SOME/IP-message. ⌋*()*

Rationale: Message-ID and Length are not covered since this allows the AUTOSAR Socket Adaptor header mode to work.

6.2.3.4 Request ID [32 Bit]

[TR_SOMEIP_00043] ⌈ The Request ID allows a client to differentiate multiple calls to the same method. Therefore, the Request ID has to be unique for a single client and server combination only. When generating a response message, the server has to copy the Request ID from the request to the response message. This allows the client to map a response to the issued request even with more than one request outstanding. ⌋*()*

[TR_SOMEIP_00044] ⌈ Request IDs might be reused as soon as the response arrived or is not expected to arrive anymore (timeout). In most automotive use cases a very low number of outstanding requests are expected. For small systems without the possibility of parallel requests, the Request ID might always set to the same value. ⌋*()*

[TR_SOMEIP_00045] ⌈ For AUTOSAR systems the Request ID needs to be structured as shown in the next section. Even for non-AUTOSAR systems it is required to encode the callers Client ID as shown in the next section. ⌋*()*

6.2.3.4.1 Structure of the Request ID

[TR_SOMEIP_00046] ⌈ In AUTOSAR the Request ID is constructed of the Client ID and Session ID:

Client ID [16 Bits]	Session ID [16 Bits]

⌋*()*

Note:
This means that the implementer of an ECU can define the Client-IDs as required by his implementation and the Server does not need to know this layout or definitions because he just copies the complete Request-ID in the response.

[TR_SOMEIP_00047] ⌈ The Client ID is the unique identifier for the calling client inside the ECU. The Session ID is a unique identifier chosen by the client for each call. If session handling is not used, the Session ID shall be set to 0x0000. ⌋*()*

[TR_SOMEIP_00532] ⌈ The Client ID shall also support being unique in the overall vehicle by having a configurable prefix or fixed value (e.g. the most significant byte of Client ID being the diagnostics address or a configured Client ID for a given application/SW-C). ⌋*()*

For example:

Client ID Prefix [8 Bits]	Client ID [8 Bits]	Session ID [16 Bits]

[TR_SOMEIP_00533] ⌈ Request/Response methods shall use session handling with Session IDs starting with `0x0001`. ⌋*()*

[TR_SOMEIP_00534] ⌈ Events, notification events, and Fire&Forget methods shall use session handling if required by the application. ⌋*()*

This could be for example because of functional safety reasons.

[TR_SOMEIP_00050] ⌈ For fire&forget methods, events and notification events the Session ID should start with 1 and be incremented for every message sent. ⌋*()*

[TR_SOMEIP_00521] ⌈ When the Session ID reaches 0xFFFF, it shall start with 0x0001 again. ⌋*()*

[TR_SOMEIP_00051] ⌈ The handling of the Request ID in SOME/IP-SD messages is discussed later in this specification. ⌋*()*

6.2.3.5 Protocol Version [8 Bit]

[TR_SOMEIP_00052] ⌈ Protocol Version is an 8 Bit field containing the SOME/IP protocol version, which currently shall be set to 0x01. ⌋*()*

6.2.3.6 Interface Version [8 Bit]

[TR_SOMEIP_00053] ⌈ Interface Version is an 8 Bit field that contains the Major Version of the Service Interface. ⌋*()*

Rationale: This is required to catch mismatches in Service definitions and allows debugging tools to identify the Service Interface used, if version is used.

6.2.3.7 Message Type [8 Bit]

[TR_SOMEIP_00055] ⌈ The Message Type field is used to differentiate different types of messages and shall contain the following values:

Number	Value	Description
`0x00`	`REQUEST`	A request expecting a response (even void)
`0x01`	`REQUEST_NO_RETURN`	A fire&forget request
`0x02`	`NOTIFICATION`	A request of a notification/event callback expecting no response
`0x40`	`REQUEST_ACK`	Acknowledgment for `REQUEST` (optional)

0x41	REQUEST_NO_RETURN_ACK	Acknowledgment for REQUEST_NO_RETURN (informational)
0x42	NOTIFICATION_ACK	Acknowledgment for NOTIFICATION (informational)
0x80	RESPONSE	The response message
0x81	ERROR	The response containing an error)
0xC0	RESPONSE_ACK	The Acknowledgment for RESPONSE (informational)
0xC1	ERROR_ACK	Acknowledgment for ERROR (informational)

⌋()

[TR_SOMEIP_00056] ⌈ Regular request (message type 0x00) will be answered by a response (message type 0x80), when no error occurred. If errors occur an error message (message type 0x81) will be sent. It is also possible to send a request that does not have a response message (message type 0x01). For updating values through notification a callback interface exists (message type 0x02). ⌋()

For all messages an optional acknowledgment (ACK) exists. These care defined for transport protocols (i.e. UDP) that do not acknowledge a received message. ACKs are only transported when the interface specification requires it. Only the usage of the REQUEST_ACK is currently specified in this document. All other ACKs are currently informational and do not need to be implemented.

6.2.3.8 Return Code [8 Bit]

[TR_SOMEIP_00058] ⌈ The Return Code is used to signal whether a request was successfully been processed. For simplification of the header layout, every message transports the field Return Code.

The Return Codes are specified in detail in [TR_SOMEIP_00191].

Messages of Type REQUEST, REQUEST_NO_RETURN, and Notification have to set the Return Code to 0x00 (E_OK). The allowed Return Codes for specific message types are:

Message Type	Allowed Return Codes
REQUEST	N/A set to 0x00 (E_OK)
REQUEST_NO_RETURN	N/A set to 0x00 (E_OK)
NOTIFICATION	N/A set to 0x00 (E_OK)
RESPONSE	See Return Codes in [TR_SOMEIP_00191]
ERROR	See Return Codes in [TR_SOMEIP_00191]. Shall not be 0x00 (E_OK).

⌋()

6.2.3.9 Payload [variable size]

[TR_SOMEIP_00060] ⌈ In the payload field the parameters are carried. The serialization of the parameters will be specified in the following section. ⌋()

[TR_SOMEIP_00061] ⌈ The size of the SOME/IP payload field depends on the transport protocol used. With UDP the SOME/IP payload shall be between 0 and 1400 Bytes. The limitation to 1400 Bytes is needed in order to allow for future changes to protocol stack (e.g. changing to IPv6 or adding security means). Since TCP supports segmentation of payloads, larger sizes are automatically supported. ⌋()

6.2.4 Serialization of Parameters and Data Structures

[TR_SOMEIP_00062] ⌈ The serialization is based on the parameter list defined by the interface specification. To allow migration of the service interface the deserialization code shall ignore parameters attached to the end of previously known parameter list; i.e. parameters that were not defined in the interface specification used to generate or parameterize the deserialization code. ⌋()

[TR_SOMEIP_00063] ⌈ The interface specification defines the exact position of all parameters in the PDU and has to consider the memory alignment. The serialization shall not try to automatically align parameters but shall be aligned as specified in the interface specification. SOME/IP payload should be placed in memory so that the SOME/IP payload is suitable aligned. For infotainment ECUs an alignment of 8 Bytes (i.e. 64 bits) should be achieved, for all ECU at least an alignment of 4 Bytes shall be achieved. ⌋()

[TR_SOMEIP_00569] ⌈ Alignment is always calculated from start of SOME/IP message. ⌋()

Note:
If a parameter has to be aligned to x Bytes, padding shall be inserted so that the relative position from start of the SOME/IP message (i.e. the position of the first header byte) modulo x equals 0.

[TR_SOMEIP_00064] ⌈ In the following the deserialization of different parameters is specified. ⌋()

6.2.4.1 Basic Datatypes

[TR_SOMEIP_00065] ⌈ The following basic datatypes shall be supported:

Type	Description	Size [bit]	Remark

boolean	TRUE/FALSE value	8	FALSE (0), TRUE (1)
uint8	unsigned Integer	8	
uint16	unsigned Integer	16	
uint32	unsigned Integer	32	
sint8	signed Integer	8	
sint16	signed Integer	16	
sint32	signed Integer	32	
float32	floating point number	32	IEEE 754 binary32 (Single Precision)
float64	floating point number	64	IEEE 754 binary64 (Double Precision)

⌋()

[TR_SOMEIP_00066] ⌈ The Byte Order is specified for each parameter by the interface definition. ⌋()

[TR_SOMEIP_00067] ⌈ In addition, uint64 and sint64 types shall be supported at least on infotainment ECUs. ⌋()

6.2.4.1.1 AUTOSAR Specifics

[TR_SOMEIP_00068] ⌈ See AUTOSAR SWS COM 7.2.2 (COM675) for supported data types. ⌋()

[TR_SOMEIP_00069] ⌈ AUTOSAR COM module shall support endianess conversion for all Integer types (COM007). ⌋()

[TR_SOMEIP_00070] ⌈ AUTOSAR defines boolean as the shortest supported unsigned Integer (Platform Types PLATFORM027). SOME/IP uses 8 Bits. ⌋()

6.2.4.2 Structured Datatypes (structs)

[TR_SOMEIP_00071] ⌈ The serialization of a struct shall be close to the in-memory layout. This means, only the parameters shall be serialized sequentially into the buffer. Especially for structs it is important to consider the correct memory alignment. Insert reserved/padding elements in the interface definition if needed for alignment, since the SOME/IP implementation shall not automatically add such padding. ⌋()

[TR_SOMEIP_00072] ⌈ So if for example a struct includes an uint8 and an uint32, they are just written sequentially into the buffer. This means that there is no padding between the uint8 and the first byte of the uint32; therefore, the uint32 might not be aligned. ⌋()

[TR_SOMEIP_00073] ⌈ If a SOME/IP generator or similar encounters an interface specification that leads to an PDU not correctly aligned (e.g. because of an unaligned struct), the SOME/IP generator shall warn about a misaligned struct but shall not fail in generating the code. ⌋*()*

[TR_SOMEIP_00074] ⌈ Warning about unaligned structs or similar shall not be done in the implementation but only in the tool chain used to generate the implementation. ⌋*()*

[TR_SOMEIP_00075] ⌈ Messages of legacy busses like CAN and FlexRay are usually not aligned. Warnings can be turned off or be ignored in such cases. ⌋*()*

[TR_SOMEIP_00076] ⌈ A struct shall be serialized exactly as specified. ⌋*()*

[TR_SOMEIP_00077] ⌈ The SOME/IP implementation shall not automatically insert dummy/padding elements. ⌋*()*

[TR_SOMEIP_00078] ⌈

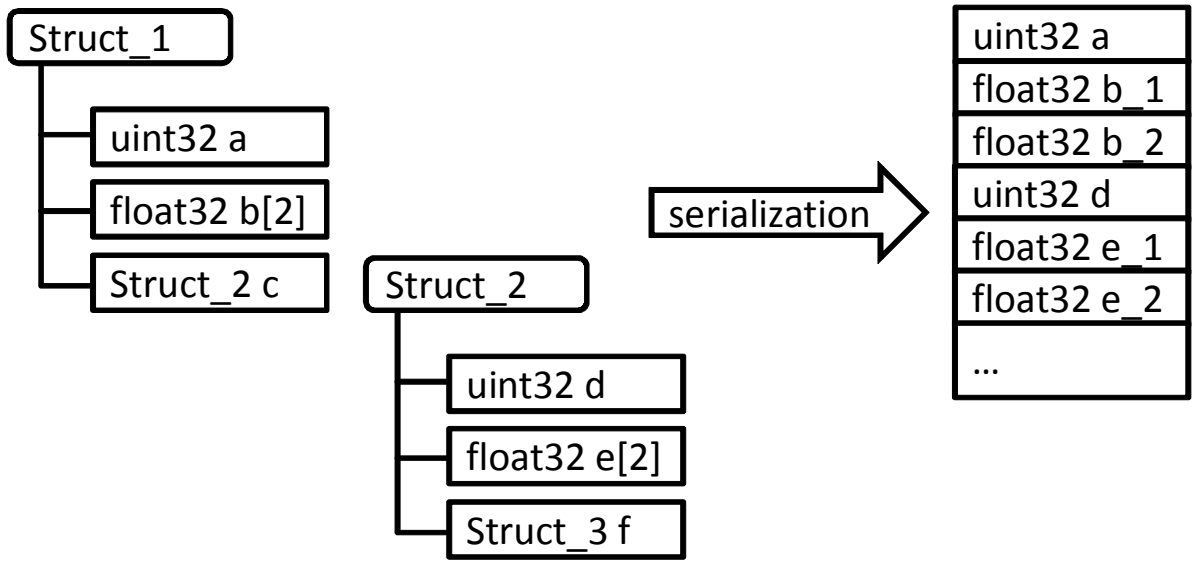

Figure 6.2: Serialization of Structs

⌋*()*

[TR_SOMEIP_00079] ⌈ The interface specification may add a length field of 8, 16 or 32 Bit in front of the Struct. ⌋*()*

[TR_SOMEIP_00080] ⌈ If the length of the length field is not specified, a length of 0 has to be assumed and no length field is in the message. ⌋*()*

[TR_SOMEIP_00081] ⌈ The length field of the struct describes the number of bytes of the struct. If the length is greater than the length of the struct as specified in the Interface Definition only the bytes specified in the Interface Specification shall be interpreted and the other bytes shall be skipped based on the length field.

This allows for extensible structs which allow better migration of interfaces. ⌋*()*

6.2.4.3 Strings (fixed length)

[TR_SOMEIP_00082] ⌈ Strings are encoded using Unicode and are terminated with a "\0"-character despite having a fixed length. The length of the string (this includes the "\0") in Bytes has to be specified in the interface definition. Fill unused space using "\0". ⌋*()*

[TR_SOMEIP_00083] ⌈ Different Unicode encoding shall be supported including UTF-8, UTF-16BE, and UTF-16LE. Since these encoding have a dynamic length of bytes per character, the maximum length in bytes is up to three times the length of characters in UTF-8 plus 1 Byte for the termination with a "\0" or two times the length of the characters in UTF-16 plus 2 Bytes for a "\0". A UTF-8 character can be up to 6 bytes and an UTF-16 character can be up to 4 bytes. ⌋*()*

[TR_SOMEIP_00084] ⌈ UTF-16LE and UTF-16BE strings shall be zero terminated with a "\0" character. This means they shall end with (at least) two 0x00 Bytes. ⌋*()*

[TR_SOMEIP_00085] ⌈ UTF-16LE and UTF-16BE strings shall have an even length. ⌋*()*

[TR_SOMEIP_00086] ⌈ For UTF-16LE and UTF-16BE strings having a odd length the last byte shall be ignored. The two bytes before shall be 0x00 bytes (termination). ⌋*()*

[TR_SOMEIP_00087] ⌈ All strings shall always start with a Byte Order Mark (BOM). The BOM shall be included in fixed-length-strings as well as dynamic-length strings. ⌋*()*

[TR_SOMEIP_00088] ⌈ The receiving SOME/IP implementation shall check the BOM against the Interface Specification and might need to handle this as an error based on this specification. ⌋*()*

[TR_SOMEIP_00089] ⌈ The BOM may be added by the application or the SOME/IP implementation. ⌋*()*

[TR_SOMEIP_00090] ⌈ The String encoding shall be specified in the interface definition. ⌋*()*

6.2.4.4 Strings (dynamic length)

[TR_SOMEIP_00091] ⌈ Strings with dynamic length start with a length field. The length is measured in Bytes and is followed by the "\0"-terminated string data. The interface definition shall also define the maximum number of bytes of the string (including termination with "\0"). ⌋*()*

[TR_SOMEIP_00092] ⌈ [TR_SOMEIP_00084], [TR_SOMEIP_00085], and [TR_SOMEIP_00086] shall also be valid for strings with dynamic length. ⌋*()*

[TR_SOMEIP_00093] ⌈ Dynamic length strings shall have a length field may of 8, 16 or, 32 Bit. This length is defined by the Interface Specification. ⌋*()*

Fixed length strings may be seen as having a 0 Bit length field.

[TR_SOMEIP_00094] ⌈ If not specified otherwise in the interface specification the length of the length field is 32 Bit (default length of length field). ⌋*()*

[TR_SOMEIP_00095] ⌈ The length of the Strings length field is not considered in the value of the length field; i.e. the length field does not count itself. ⌋*()*

[TR_SOMEIP_00096] ⌈ Supported encodings are defined as in Chapter 6.2.4.3. ⌋*()*

[TR_SOMEIP_00097] ⌈ If the interface definition states the alignment of the next data element the string shall be extended with "\0" characters to meet the alignment. ⌋*()*

6.2.4.5 Arrays (fixed length)

[TR_SOMEIP_00098] ⌈ The length of fixed length arrays is defined by the interface definition. ⌋*()*

They can be seen as repeated elements. In chapter 6.2.4.7 dynamic length arrays are shown, which can be also used. Fixed length arrays are easier for use in very small devices. Dynamic length arrays might need more resources on the ECU using them.

6.2.4.5.1 One-dimensional

[TR_SOMEIP_00099] ⌈ The one-dimensional arrays with fixed length n carry exactly n elements of the same type. The layout is shown in Figure 6.3. ⌋*()*

[TR_SOMEIP_00100] ⌈

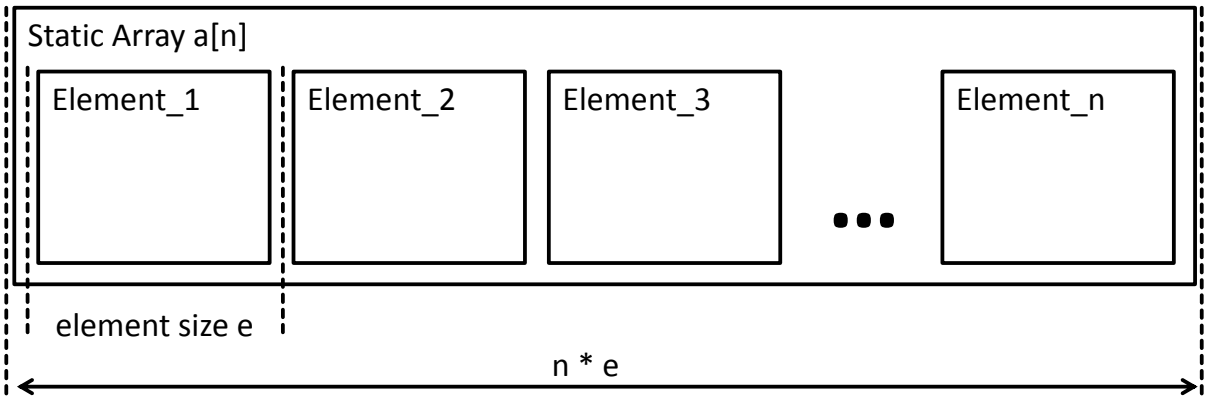

Figure 6.3: One-dimensional array (fixed length)

⌋*()*

6.2.4.5.2 Multidimensional

[TR_SOMEIP_00101] ⌈ The serialization of multidimensional arrays follows the in-memory layout of multidimensional arrays in the C++ programming language (row-major order) and is shown in Figure 6.4. ⌋()

[TR_SOMEIP_00102] ⌈

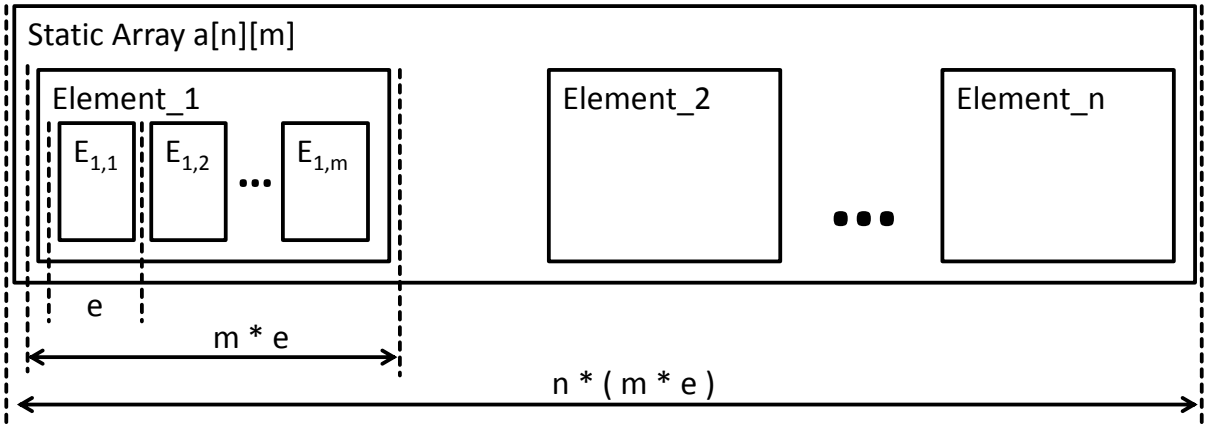

Figure 6.4: Multidimensional array (fixed length)

⌋()

6.2.4.5.3 AUTOSAR Specifics

[TR_SOMEIP_00103] ⌈ Consult AUTOSAR SWS RTE chapter 5.3.4.4 for Arrays. ⌋()

6.2.4.6 Optional Parameters / Optional Elements

[TR_SOMEIP_00105] ⌈ Optional Elements shall be encoded as array with 0 to 1 elements. For the serialization of arrays with dynamic length see Chapter 6.2.4.7. ⌋()

6.2.4.7 Dynamic Length Arrays

[TR_SOMEIP_00106] ⌈ The layout of arrays with dynamic length basically is based on the layout of fixed length arrays. To determine the size of the array the serialization adds a length field (default length 32 bit) in front of the data, which counts the bytes of the array. The length does not include the size of the length field. Thus, when transporting an array with zero elements the length is set to zero. ⌋()

[TR_SOMEIP_00107] ⌈ The Interface Definition may define the length of the length field. Length of 0, 8, 16, and 32bit are allowed. If the length of the length field is set to 0 Bits, the number of elements in the array has to be fixed; thus, being an array with fixed length. ⌋()

[TR_SOMEIP_00108] ⌈ The layout of dynamic arrays is shown in 6.5 and Figure 6.6. ⌋()

[TR_SOMEIP_00109] ⌈

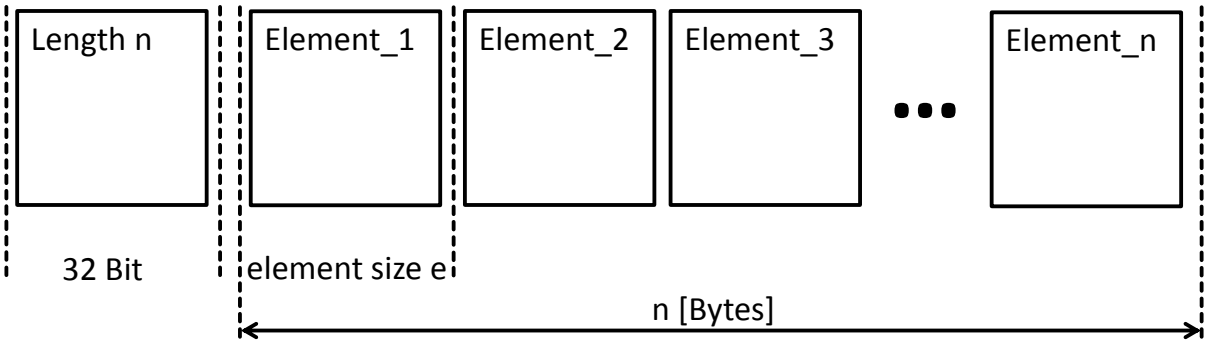

Figure 6.5: One-dimensional array (dynamic length)

⌋()

[TR_SOMEIP_00110] ⌈ In the one-dimensional array one length field is used, which carries the number of bytes used for the array. ⌋()

[TR_SOMEIP_00111] ⌈ The number of static length elements can be easily calculated by dividing by the size of an element. ⌋()

[TR_SOMEIP_00112] ⌈ In the case of dynamical length elements the number of elements cannot be calculated but the elements must be parsed sequentially. ⌋()

[TR_SOMEIP_00113] ⌈

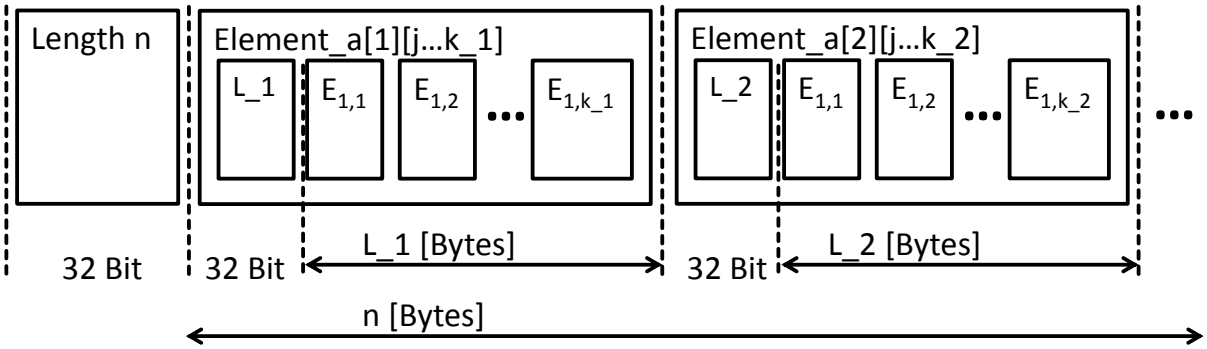

Figure 6.6: Multidimensional array (dynamic length)

⌋()

[TR_SOMEIP_00114] ⌈ In multidimensional arrays multiple length fields are needed. ⌋()

[TR_SOMEIP_00115] ⌈ If static buffer size allocation is required, the interface definition shall define the maximal length of each dimension. ⌋()

Rationale: When measuring the length in Bytes, complex multi-dimensional arrays can be skipped over in deserialization.

[TR_SOMEIP_00115] even supports that different length columns and different length rows in the same dimension. See `k_1` and `k_2` in Figure 6.6.

6.2.4.8 Enumeration

[TR_SOMEIP_00117] ⌈ The interface definition might specify an enumeration based on unsigned integer datatypes (uint8, uint16, uint32, uint64). ⌋*()*

6.2.4.9 Bitfield

[TR_SOMEIP_00300] ⌈ Bitfields shall be transported as basic datatypes uint8/uint16/uint32. ⌋*()*

[TR_SOMEIP_00301] ⌈ The interface definition shall be able to define the name of each bit. ⌋*()*

[TR_SOMEIP_00302] ⌈ The interface definition shall be able to define the names of the values a bit can be set to. ⌋*()*

Each SOME/IP implementation may choose to de/serialize a bitfield or hand up the uint8/uint16/uint32 to the application. A SOME/IP implementation may allow turning the de/serialization of a bitfield on or off.

6.2.4.10 Union / Variant

[TR_SOMEIP_00118] ⌈ A union (also called variant) is a parameter that can contain different types of elements. For example, if one defines a union of type uint8 and type uint16, the union shall carry an element of uint8 or uint16. ⌋*()*

It is clear that that when using different types of elements the alignment of subsequent parameters may be distorted. To resolve this, padding might be needed.

[TR_SOMEIP_00119] ⌈ The default serialization layout of unions in SOME/IP is as follows:

Length field [32 bit]
Type field [32 bit]
Element including padding [sizeof(padding) = length - sizeof(element)]

⌋*()*

[TR_SOMEIP_00120] ⌈ The order of the length and type field is adjustable by the interface specification. If this is not specified the default layout as in [TR_SOMEIP_00119] shall be used. ⌋*()*

[TR_SOMEIP_00121] ⌈ The length of the length field shall be defined by the Interface Specification and shall be 32, 16, 8, or 0 bits ⌋*()*

[TR_SOMEIP_00122] ⌈ A length of the length field of 0 Bit means that no length field will be written to the PDU. ⌋*()*

[TR_SOMEIP_00123] ⌈ If the length of the length field is 0 Bit, all types in the union shall be of the same length. ⌋*()*

[TR_SOMEIP_00124] ⌈ If the interface specification defines a union with a length field of 0 Bits and types with different length, a SOME/IP implementation shall warn about this and use the length of the longest element and pad all others with zeros (0x00). ⌋*()*

[TR_SOMEIP_00125] ⌈ If the Interface Specification does not specify the length of the length field for a union, 32 bit length of the length field shall be used. ⌋*()*

[TR_SOMEIP_00126] ⌈ The length field defines the size of the element and padding in bytes and does not include the size of the length field and type field. ⌋*()*

[TR_SOMEIP_00127] ⌈ The length of the type field shall be defined by the Interface Specification and shall be 32, 16, or 8 bits. ⌋*()*

[TR_SOMEIP_00128] ⌈ If the Interface Specification does not specify the length of the type field of a union, 32 bit length of the type field shall be used. ⌋*()*

[TR_SOMEIP_00129] ⌈ The type field describes the type of the element. Possible values of the type field are defined by the interface specification for each union separately. The types are encoded as in the interface specification in ascending order starting with 1. The 0 is reserved for the NULL type - i.e. an empty union. The Interface Definition shall allow or disallow the usage of NULL for a Union/Variant. ⌋*()*

[TR_SOMEIP_00130] ⌈ The element is serialized depending on the type in the type field. This also defines the length of the data. All bytes behind the data that are covered by the length, are padding. The deserializer shall skip such bytes according to the length field. The value of the length field for each type shall be defined by the interface specification. ⌋*()*

[TR_SOMEIP_00131] ⌈ By using a struct different padding layouts can be achieved. ⌋*()*

6.2.4.10.1 Example: Union of uint8/uint16 both padded to 32 bit

[TR_SOMEIP_00132] ⌈ In this example a length of the length field is specified as 32 Bits. The union shall support a uint8 and a uint16 as elements. Both are padded to the 32 bit boundary (length=4 Bytes). ⌋*()*

[TR_SOMEIP_00133] ⌈ A uint8 will be serialized like this:

Length = 4 Bytes
Type = 1

uint8	Padding 0x00	Padding 0x00	Padding 0x00

⌋()

[TR_SOMEIP_00134] ⌈ A uint16 will be serialized like this:

Length = 4 Bytes		
Type = 2		
uint16	Padding 0x00	Padding 0x00

⌋()

6.2.4.11 Example Map / Dictionary

[TR_SOMEIP_00135] ⌈ Maps or dictionaries can be easily described as an array of key-value-pairs. The most basic way to implement a map or dictionary would be an array of a struct with two fields: key and value. Since the struct has no length field, this is as efficient as a special map or dictionary type could be. When choosing key and value as uint16, a serialized map with 3 entries looks like this:

Length = 12 Bytes	
key0	value0
key1	value1
key2	value2

⌋()

6.3 RPC Protocol specification

This chapter describes the RPC protocol of SOME/IP.

6.3.1 Transport Protocol Bindings

[TR_SOMEIP_00137] ⌈ In order to transport SOME/IP messages of IP different transport protocols may be used. SOME/IP currently supports UDP and TCP. Their bindings are explained in the following sections, while Chapter 6.4 discusses which transport protocol to choose. ⌋()

[TR_SOMEIP_00138] ⌈ If a server runs different instances of the same service, messages belonging to different service instances shall be mapped to the service instance by the transport protocol port on the server side. ⌋()

[TR_SOMEIP_00535] ⌈ All Transport Protocol Bindings shall support transporting more than one SOME/IP message in a Transport Layer PDU (i.e. UDP packet or TCP segment). ⌋*()*

[TR_SOMEIP_00142] ⌈ The receiving SOME/IP implementation shall be capable of receiving unaligned SOME/IP messages transported by UDP or TCP. ⌋*()*

Rationale:
When transporting multiple SOME/IP payloads in UDP or TCP the alignment of the payloads can be only guaranteed, if the length of every payloads is a multiple of the alignment size (e.g. 32 bits).

6.3.1.1 UDP Binding

[TR_SOMEIP_00139] ⌈ The UDP binding of SOME/IP is straight forward by transporting SOME/IP messages in UDP packets. The SOME/IP messages shall not be fragmented. Therefore care shall be taken that SOME/IP messages are not too big, i.e. up to 1400 Bytes of SOME/IP payload. Messages with bigger payload shall not be transported over UDP but with e.g. TCP. ⌋*()*

[TR_SOMEIP_00140] ⌈ The header format allows transporting more than one SOME/IP message in a single UDP packet. The SOME/IP implementation shall identify the end of a SOME/IP message by means of the SOME/IP length field. Based on the UDP length field, SOME/IP shall determine if there are additional SOME/IP messages in the UDP packet. ⌋*()*

[TR_SOMEIP_00141] ⌈ Each SOME/IP payload shall have its own SOME/IP header. ⌋*()*

6.3.1.1.1 AUTOSAR specific

[TR_SOMEIP_00145] ⌈ Based on the Socket Adaptor concept AUTOSAR shall divide an incoming UDP packet into different I-PDUs. However, not all AUTOSAR implementations are currently able to combine different I-PDUs and send an UDP-Packet with more than one SOME/IP message. ⌋*()*

6.3.1.2 TCP Binding

[TR_SOMEIP_00146] ⌈ The TCP binding of SOME/IP is heavily based on the UDP binding. In contrast to the UDP binding, the TCP binding allows much bigger SOME/IP messages and uses the robustness features of TCP (coping with loss, reorder, duplication, etc.). ⌋*()*

[TR_SOMEIP_00147] ⌈ Every SOME/IP payload shall have its own SOME/IP header. ⌋*()*

[TR_SOMEIP_00148] ⌈ In order to lower latency and reaction time, Nagle's algorithm shall be turned off (TCP_NODELAY). ⌋*()*

[TR_SOMEIP_00149] ⌈ When the TCP connection is lost, outstanding requests shall be handled as timeouts. Since TCP handles reliability, additional means of reliability are not needed. Error handling is discussed in detail in Chapter 6.3.6. ⌋*()*

[TR_SOMEIP_00150] ⌈ The client and server shall use a single TCP connection for all methods, events, and notifications of a service instance. When having more than one instance of a service a TCP connection per services instance is needed. ⌋*()*

[TR_SOMEIP_00151] ⌈ The TCP connection shall be used for as many services as possible to minimize the number of TCP connections between the client and the server (basically only one connection per TCP port of server). ⌋*()*

[TR_SOMEIP_00152] ⌈ The TCP connection shall be opened by the client, when the first method call shall be transport or the client tries to receive the first notifications. ⌋*()*

[TR_SOMEIP_00153] ⌈ The client is responsible for reestablishing the TCP connection whenever it fails. ⌋*()*

[TR_SOMEIP_00522] ⌈ The TCP connection shall be closed by the client, when the TCP connection is not required anymore. ⌋*()*

[TR_SOMEIP_00523] ⌈ The TCP connection shall be closed by the client, when all Services using the TCP connections are not available anymore (stopped or timed out). ⌋*()*

[TR_SOMEIP_00524] ⌈ The server shall not stop the TCP connection when stopping all services the TCP connection was used for but give the client enough time to process the control data to shutdown the TCP connection itself. ⌋*()*

Rational:
When the server closes the TCP connection before the client recognized that the TCP is not needed anymore, the client will try to reestablish the TCP connection.

6.3.1.2.1 Allowing resync to TCP stream using Magic Cookies

[TR_SOMEIP_00154] ⌈ In order to allow resynchronization to SOME/IP over TCP in testing and integration scenarios the SOME/IP Magic Cookie Message (Figure 6.7) shall be used between SOME/IP messages over TCP. ⌋*()*

[TR_SOMEIP_00155] ⌈ Each TCP segment shall start with a SOME/IP Magic Cookie Message. ⌋*()*

[TR_SOMEIP_00156] ⌈ The implementation shall only include up to one SOME/IP Magic Cookie Message per TCP segment. ⌋*()*

[TR_SOMEIP_00157] ⌈ If the implementation has no appropriate access to the message segmentation mechanisms and therefore cannot fulfill [TR_SOMEIP_00155] and

[TR_SOMEIP_00156], the implementation shall include SOME/IP Magic Cookie Messages based on the following heuristic: ⌋()

[TR_SOMEIP_00158] ⌈ Add SOME/IP Magic Cookie Message once every 10 seconds to the TCP connection as long as messages are transmitted over this TCP connection. ⌋()

[TR_SOMEIP_00159] ⌈ The layout of the Magic Cookie Message is based on SOME/IP. The fields are set as follows:

- Service ID = 0xFFFF
- Method ID = 0x0000 (for the direction Client to Server)
- Method ID = 0x8000 (for the direction Server to Client)
- Length = 0x0000 0008
- Client ID = 0xDEAD
- Session ID = 0xBEEF
- Protocol Version as specified above ([TR_SOMEIP_00052])
- Interface Version = 0x01
- Message Type = 0x01 (for Client to Server Communication) or 0x02 (for Server to Client Communication)
- Return Code = 0x00

⌋()

[TR_SOMEIP_00160] ⌈ The layout of the Magic Cookie Messages is shown in Figure 6.7. ⌋()

[TR_SOMEIP_00161] ⌈

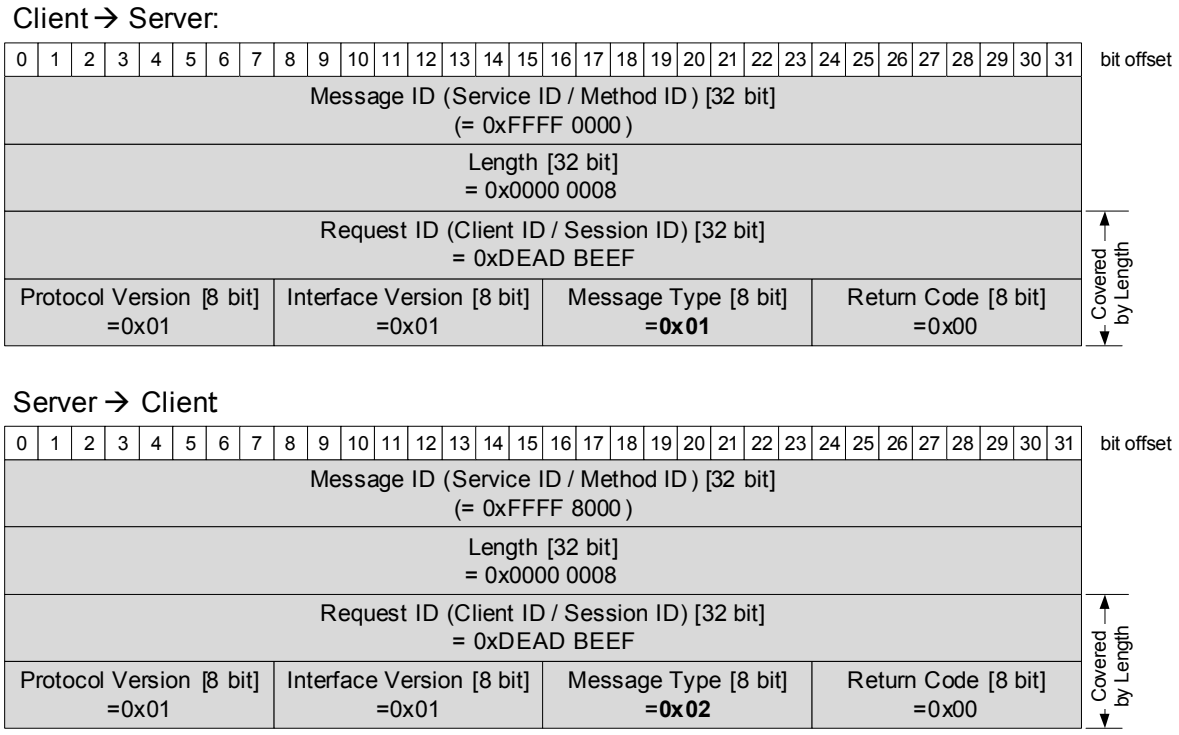

Figure 6.7: SOME/IP Magic Cookie Message for SOME/IP

⌋()

6.3.1.3 Multiple Service-Instances

[TR_SOMEIP_00162] ⌈ Service-Instances of the same Service are identified through different Instance IDs. It shall be supported that multiple Service-Instances reside on different ECUs as well as multiple Service-Instances of one or more Services reside on one single ECU. ⌋()

[TR_SOMEIP_00163] ⌈ While different Services shall be able to share the same port number of the transport layer protocol used, multiple Service-Instances of the same service on one single ECU shall listen on different ports per Service-Instance. ⌋()

Rationale: While Instance IDs are used for Service Discovery, they are not contained in the SOME/IP header.

[TR_SOMEIP_00165] ⌈ A Service Instance can be identified through the combination of the Service ID combined with the socket (i.e. IP-address, transport protocol (UDP/TCP), and port number). It is recommended that instances use the same port number for UDP and TCP. If a service instance uses UDP port x, only this instance of the service and not another instance of the same service should use exactly TCP port x for its services. ⌋()

6.3.2 Request/Response Communication

[TR_SOMEIP_00166] ⌈ One of the most common communication patterns is the request/response pattern. One communication partner (in the following called the client) sends a request message, which is answered by another communication partner (the server). ⌋*()*

[TR_SOMEIP_00167] ⌈ For the SOME/IP request message the client has to do the following for payload and header:

- Construct the payload
- Set the Message ID based on the method the client wants to call
- Set the Length field to 8 bytes (for the part of the SOME/IP header after the length field) + length of the serialized payload
- Optionally set the Request ID to a unique number (shall be unique for client only)
- Set the Protocol Version according [TR_SOMEIP_00052]
- Set the Interface Version according to the interface definition
- Set the Message Type to Request (i.e. 0x00)
- Set the Return Code to 0x00

⌋*()*

[TR_SOMEIP_00168] ⌈ The server builds it header based on the header of the client and does in addition:

- Construct the payload
- Set the length to the 8 Bytes + new payload size
- Set the Message Type to RESPONSE (i.e. 0x80) or ERROR (i.e. 0x81)
- Set the Return Code.

⌋*()*

6.3.2.1 AUTOSAR Specific

[TR_SOMEIP_00169] ⌈ AUTOSAR may implement Request-Response by means of the Client/Server-Functionality. For some implementations it might be necessary to implement the inter-ECU communication by means of the Sender/Receiver-Functionality. In this case the semantics and syntax of the PDU shall not differ to this specification. ⌋*()*

6.3.3 Fire&Forget Communication

[TR_SOMEIP_00170] ⌈ Requests without response message are called fire&forget. The implementation is basically the same as for Request/Response with the following differences:

- There is no response message.
- The message type is set to REQUEST_NO_RETURN (i.e. 0x01)

⌋()

[TR_SOMEIP_00171] ⌈ Fire & Forget messages shall not return an error. Error handling and return codes shall be implemented by the application when needed. ⌋()

6.3.3.1 AUTOSAR Specific

[TR_SOMEIP_00172] ⌈ fire&forget should be implemented using the Sender/Receiver-Functionality. ⌋()

6.3.4 Notification Events

[TR_SOMEIP_00173] ⌈ Notifications describe a general Publish/Subscribe-Concept. Usually the server publishes a service to which a client subscribes. On certain events the server will send the client a event, which could be for example an updated value or an event that occurred. ⌋()

[TR_SOMEIP_00174] ⌈ SOME/IP is used only for transporting the updated value and not for the publishing and subscription mechanisms. These mechanisms are implemented by SOME/IP-SD and are explained in Chapter 6.3.4.2. ⌋()

[TR_SOMEIP_00175] ⌈ When more than one subscribed client on the same ECU exists, the system shall handle the replication of notifications in order to save transmissions on the communication medium. This is especially important, when notifications are transported using multicast messages. ⌋()

6.3.4.1 Strategy for sending notifications

[TR_SOMEIP_00176] ⌈ For different use cases different strategies for sending notifications are possible and shall be defined in the service interface. The following examples are common:

- Cyclic update — send an updated value in a fixed interval (e.g. every 100 ms for safety relevant messages with Alive)

- Update on change — send an update as soon as a "value" changes (e.g. door open)
- Epsilon change — only send an update when the difference to the last value is greater than a certain epsilon. This concept may be adaptive, i.e. the prediction is based on a history; thus, only when the difference between prediction and current value is greater than epsilon an update is transmitted.

⌋()

6.3.4.2 Publish/Subscribe Handling

[TR_SOMEIP_00177] ⌈ Publish/Subscribe handling shall be implemented according to Chapter 6.7.7. ⌋()

6.3.4.3 AUTOSAR Specific

[TR_SOMEIP_00178] ⌈ Notifications are transported with AUTOSAR Sender/Receiver-Functionality. In case of different notification receivers within an ECU, the replication of notification messages can be done for example in the RTE. This means a event/notification message shall be only sent once to ECU with multiple recipients. ⌋()

6.3.5 Fields

[TR_SOMEIP_00179] ⌈ A field shall be a combination of getter, setter and notification event. ⌋()

[TR_SOMEIP_00180] ⌈ A field without a setter and without a getter and without a notifier shall not exist. The field shall contain at least a getter, a setter, or a notifier. ⌋()

[TR_SOMEIP_00181] ⌈ The getter of a field shall be a request/response call that has an empty payload in the request message and the value of the field in the payload of the response message. ⌋()

[TR_SOMEIP_00182] ⌈ The setter of a field shall be a request/response call that has the desired value of the field in the payload of the request message and the value that was set to the field in the payload of the response message. ⌋()

Note:
If the value of the request payload was adapted (e.g. because it was out of limits) the adapted value will be transported in the response payload.

[TR_SOMEIP_00183] ⌈ The notifier shall send an event message that transports the value of a field on change and follows the rules for events. ⌋()

6.3.6 Error Handling

[TR_SOMEIP_00184] ⌈ The error handling can be done in the application or the communication layer below. Therefore different possible mechanisms exist. ⌋*()*

6.3.6.1 Transporting Application Error Codes and Exceptions

[TR_SOMEIP_00185] ⌈ For the error handling two different mechanisms are supported. All messages have a return code field to carry the return code. However, only responses (Message Types 0x80 and 0x81) use this field to carry a return code to the request (Message Type 0x00) they answer. All other messages set this field to 0x00 (see Chapter 6.2.3.7). For more detailed errors the layout of the Error Message (Message Type 0x81) can carry specific fields for error handling, e.g. an Exception String. Error Messages are sent instead of Response Messages. ⌋*()*

[TR_SOMEIP_00186] ⌈ This can be used to handle all different application errors that might occur in the server. In addition, problems with the communication medium or intermediate components (e.g. switches) may occur, which have to be handled e.g. by means of reliable transport. ⌋*()*

6.3.6.2 Return Code

[TR_SOMEIP_00187] ⌈ The Error Handling is based on an 8 Bit Std_returnType of AUTOSAR. The two most significant bits are reserved and shall be set to 0. The receiver of a return code shall ignore the values of the two most significant bits. ⌋*()*

[TR_SOMEIP_00188] ⌈ The system shall not return an error message for events/notifications. ⌋*()*

[TR_SOMEIP_00189] ⌈ The system shall not return an error message for fire&forget methods. ⌋*()*

[TR_SOMEIP_00537] ⌈ The system shall not return an error message for events/notifications and fire&forget methods if the Message Type is set incorrectly to Request or Response. ⌋*()*

[TR_SOMEIP_00190] ⌈ For request/response methods the error message shall copy over the fields of the SOME/IP header (i.e. Message ID, Request ID, and Interface Version) but not the payload. In addition Message Type and Return Code have to be set to the appropriate values. ⌋*()*

[TR_SOMEIP_00538] ⌈ The SOME/IP implementation shall not use an unknown protocol version but write a supported protocol version in the header. ⌋*()*

[TR_SOMEIP_00191] ⌈ The following Return Codes are currently defined and shall be implemented as described:

ID	Name	Description
0x00	E_OK	No error occurred
0x01	E_NOT_OK	An unspecified error occurred
0x02	E_UNKNOWN_SERVICE	The requested Service ID is unknown.
0x03	E_UNKNOWN_METHOD	The requested Method ID is unknown. Service ID is known.
0x04	E_NOT_READY	Service ID and Method ID are known. Application not running.
0x05	E_NOT_REACHABLE	System running the service is not reachable (internal error code only).
0x06	E_TIMEOUT	A timeout occurred (internal error code only).
0x07	E_WRONG_PROTOCOL_VERSION	Version of SOME/IP protocol not supported
0x08	E_WRONG_INTERFACE_VERSION	Interface version mismatch
0x09	E_MALFORMED_MESSAGE	Deserialization error, so that payload cannot be deserialized.
0x0a	E_WRONG_MESSAGE_TYPE	An unexpected message type was received (e.g. REQUEST_NO_RETURN for a method defined as REQUEST.)
0x0b - 0x1f	RESERVED	Reserved for generic SOME/IP errors. These errors will be specified in future versions of this document.
0x20 - 0x3f	RESERVED	Reserved for specific errors of services and methods. These errors are specified by the interface specification.

⌋()

[TR_SOMEIP_00192] ⌈ Generation and handling of return codes shall be configurable. ⌋()

[TR_SOMEIP_00575] ⌈ The SOME/IP message shall be checked in the following order:

- Protocol Version supported?
- Service ID supported?
- Interface Version of this service supported?
- Method ID supported?
- Message Type supported?
- Message Type as specified in IDL?

- Payload parseable?

⌋()

[TR_SOMEIP_00539] ⌈ Implementations shall not answer with errors to SOME/IP message already carrying an error (i.e. return code `0x01` - `0x1f`). ⌋()

6.3.6.3 Error Message Format

[TR_SOMEIP_00194] ⌈ For a more flexible error handling, SOME/IP allows the user to specify a message layout specific for errors instead of using the message layout for response messages. This is defined by the interface specification and can be used to transport exceptions of higher level programming languages. ⌋()

[TR_SOMEIP_00195] ⌈ The recommended layout for the exception message is the following:

- Union of specific exceptions. At least a generic exception without fields needs to exist.
- Dynamic Length String for exception description.

⌋()

[TR_SOMEIP_00196] ⌈ The union gives the flexibility to add new exceptions in the future in a type-safe manner. The string is used to transport human readable exception descriptions to ease testing and debugging. ⌋()

6.3.6.4 Error Processing Overview

The error handling of SOME/IP is shown as an example flow chart in Figure 6.8. This does not include the application based error handling but just covers the error handling in messaging and RPC.

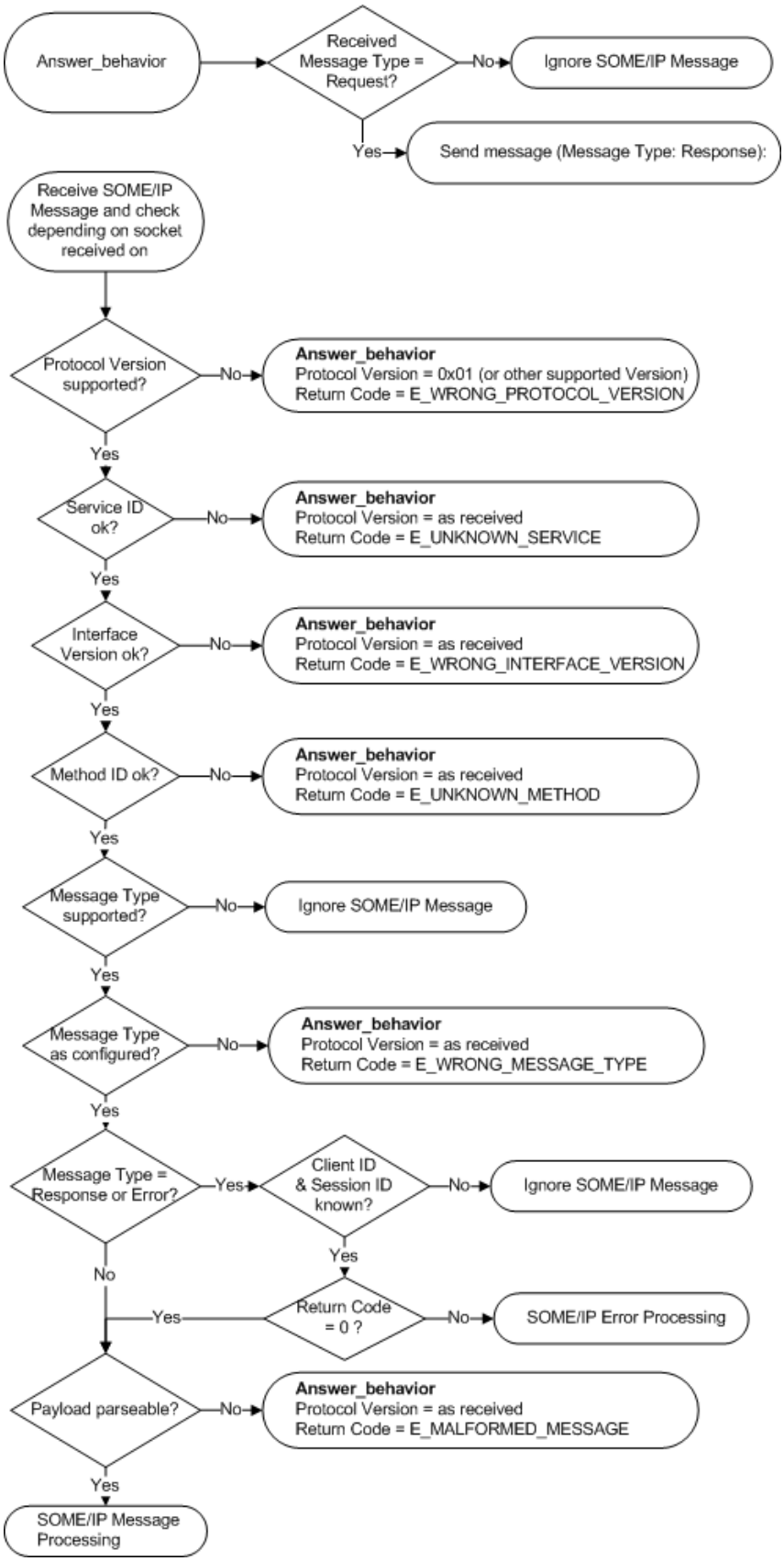

Figure 6.8: Error Handling in SOME/IP

[TR_SOMEIP_00576] ⌈ Error handling shall be based on the message type received (e.g. only methods can be answered with a return code) and shall be checked in a defined order of [TR_SOMEIP_00575]. ⌋()

6.3.6.5 Communication Errors and Handling of Communication Errors

[TR_SOMEIP_00197] ⌈ When considering the transport of RPC messages different reliability semantics exist:

- Maybe — the message might reach the communication partner
- At least once — the message reaches the communication partner at least once
- Exactly once — the message reaches the communication partner exactly once

⌋()

[TR_SOMEIP_00198] ⌈ When using these terms in regard to Request/Response the term applies to both messages (i.e. request and response or error). ⌋()

[TR_SOMEIP_00199] ⌈ While different implementations may implement different approaches, SOME/IP currently achieves "maybe" reliability when using the UDP binding and "exactly once" reliability when using the TCP binding. Further error handling is left to the application. ⌋()

[TR_SOMEIP_00200] ⌈ For "maybe" reliability, only a single timeout is needed, when using request/response communication in combination of UDP as transport protocol. Figure 6.9 shows the state machines for "maybe" reliability. The client's SOME/IP implementation has to wait for the response for a specified timeout. If the timeout occurs SOME/IP shall signal E_TIMEOUT to the client application. ⌋()

[TR_SOMEIP_00201] ⌈

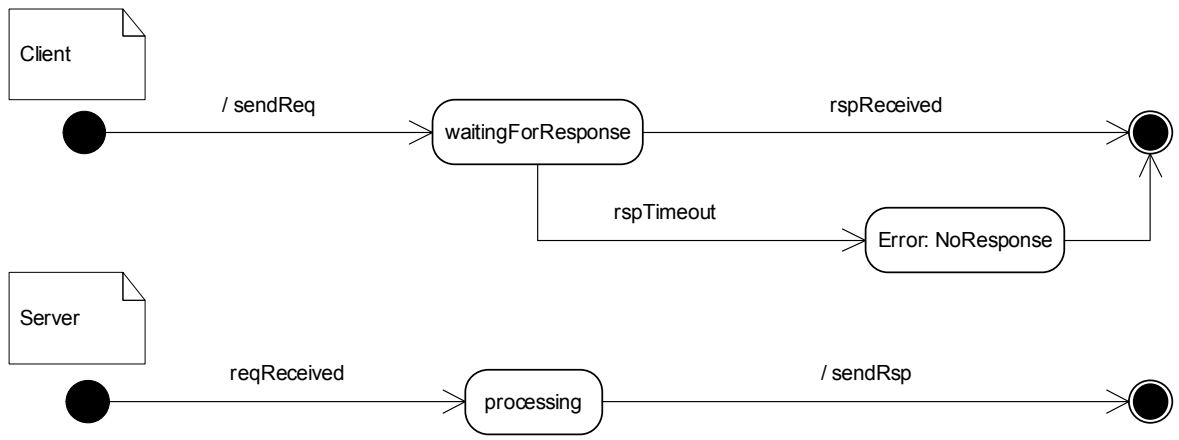

Figure 6.9: State Machines for Reliability "Maybe"

⌋()

[TR_SOMEIP_00202] ⌈ For "exactly once" reliability the TCP binding may be used, since TCP was defined to allow for reliable communication. ⌋*()*

[TR_SOMEIP_00203] ⌈ Additional mechanisms to reach higher reliability may be implemented in the application or in a SOME/IP implementation. Keep in mind that the communication does not have to implement these features. Chapter 6.3.6.5.1 describes such optional reliability mechanisms. ⌋*()*

6.3.6.5.1 Application based Error Handling

[TR_SOMEIP_00204] ⌈ The application can easily implement "at least once" reliability by using idempotent operations (i.e. operation that can be executed multiple times without side effects) and using a simple timeout mechanism. Figure 6.10 shows the state machines for "at least once" reliability using implicit acknowledgements. When the client sends out the request it starts a timer with the timeout specified for the specific method. If no response is received before the timer expires (round transition at the top), the client will retry the operation. A Typical number of retries would be 2, so that 3 requests are sent. ⌋*()*

[TR_SOMEIP_00205] ⌈ The number of retries, the timeout values, and the timeout behavior (constant or exponential back off) are outside of the SOME/IP specification and may be added to the interface definition. ⌋*()*

[TR_SOMEIP_00206] ⌈

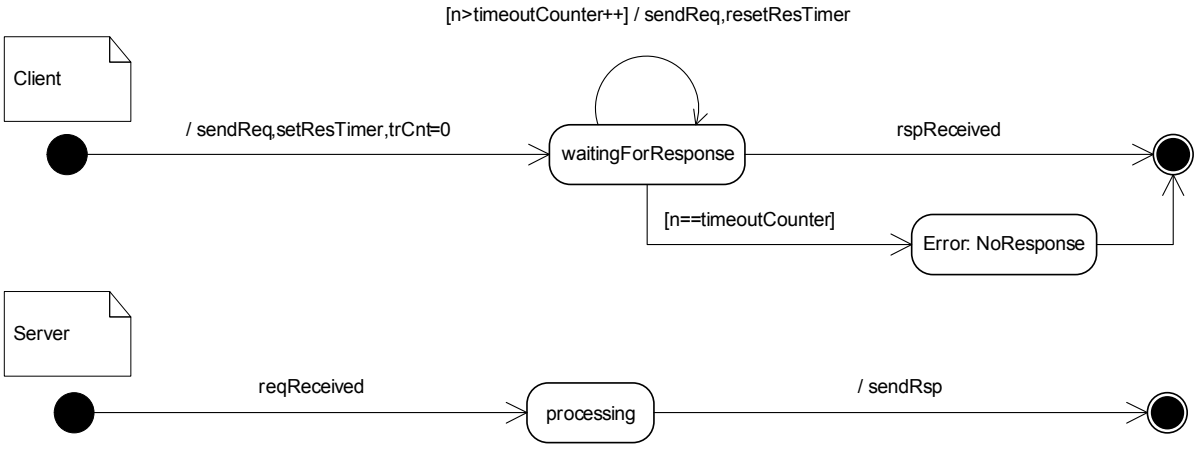

Figure 6.10: State Machines for Reliability "At least once" (idempotent operations)

⌋*()*

6.4 Guidelines on SOME/IP

6.4.1 Choosing the transport protocol

[TR_SOMEIP_00207] ⌈ SOME/IP directly supports the two most used transport protocols of the Internet: User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). While UDP is a very lean transport protocol supporting only the most important features (multiplexing and error detecting using a checksum), TCP adds additional features for achieving a reliable communication. TCP can not only handle bit errors but also segmentation, loss, duplication, reordering, and network congestion; thus, TCP is the more powerful transport protocol. ⌋*()*

[TR_SOMEIP_00208] ⌈ For use inside the vehicle, requirements are not the same as for the Internet. For many applications, we require a very short timeout to react in a very short time. These requirements are better met using UDP because the application itself can handle the unlikely event of errors. For example, in use cases with cyclic data it is often the best approach to just wait for the next data transmission instead of trying to repair the last one. The major disadvantage of UDP is that it does not handle segmentation; thus, only being able to transport smaller chunks of data. ⌋*()*

[TR_SOMEIP_00209] ⌈ Guideline:

- Use UDP if very hard latency requirements (<100ms) in case of errors is needed
- Use TCP only if very large chunks of data need to be transported (> 1400 Bytes) and no hard latency requirements in the case of errors exists
- Try using external transport or transfer mechanisms (Network File System, APIX link, 1722, ...) when more suited for the use case. In this case SOME/IP can transport a file handle or a comparable identifier. This gives the designer additional freedom (e.g. in regard to caching).

⌋*()*

[TR_SOMEIP_00210] ⌈ The transport protocol used is specified by the interface specification on a per-message basis. Methods, Events, and Fields should commonly only use a single transport protocol. ⌋*()*

6.4.2 Implementing Advanced Features in AUTOSAR Applications

[TR_SOMEIP_00211] ⌈ Unfortunately, not all features of SOME/IP can be directly supported within this release of AUTOSAR (e.g. dynamic length arrays). In the uncommon case that an advanced feature is needed within an AUTOSAR implementation not supporting it directly, a solution exists: The advanced feature shall be implemented inside the application by passing the SOME/IP payload or parts of it by means of a uint8 buffer through AUTOSAR. For AUTOSAR the fields seem to be just a dynamic length uint8 array and shall be configured accordingly. ⌋*()*

6.4.3 Serialization of Data Structures Containing Pointers

[TR_SOMEIP_00213] ⌈ For the serialization of data structures containing pointers (e.g. a tree in memory), the pointers cannot be just transferred using a data type (e.g. uint8) but shall be converted for transport. Different approaches for the serialization of pointers exist. We recommend the following approaches. ⌋*()*

6.4.3.1 Array of data structures with implicit ID

[TR_SOMEIP_00214] ⌈ When transporting a set of data structures with pointers that is small enough to fit into a single RPC message:

- Store data structures (e.g. tree nodes) in array
- Use position in array as ID of stored data structure
- Replace pointers with IDs of the data structures pointed to

⌋*()*

6.4.3.2 Array of data structures with explicit ID

[TR_SOMEIP_00215] ⌈ With larger sets of data structures additional problems have to be resolved. Since not all data structures fit into a single message the IDs have to be unique over different messages. This can be achieved in different ways:

- Add an offset field to every message. The ID of an array element will be calculated by adding the offset to its position in the array. Keep in mind that the offset needs to be carefully chosen. If for example every message can contain up to ten data structures (0-9), the offset could be chosen as 0, 10, 20, 30, and so on.
- Store an explicit ID by using an array of structs. The first field in the struct will be an ID (e.g. uint32) and the second field the data structure itself. For security and reliability reasons the pointer (i.e. the memory address) should never be used directly as ID.

⌋*()*

6.5 Compatibility rules for Interface Design (informational)

[TR_SOMEIP_00216] ⌈ As for all serialization formats migration towards a newer service interface is limited. Using a set of compatibility rules, SOME/IP allows for evolution of service interfaces. One can do the following additions and enhancements in a non-breaking way:

- Add new method to a service

  - Shall be implemented at server first.
- Add parameter to the end of a method's in or out parameters
  - When the receiver adds them first, a default value has to be defined
  - When the sender adds them first, the receiver will ignore them
- Add an exception to the list of exceptions a method can throw
- Should update client first
- If exception is unknown, "unknown exception" needs to be thrown. The exception description string however can be used
- Add new type to union
  - Should update receiver first
  - Can be skipped if unknown (sender updates first)
- Define a new data type for new methods
- Define a new exception for new methods

⌋()

[TR_SOMEIP_00217] ⌈ Not all of these changes can be introduced at the client or server side first. In some cases only the client or server can be changed first. For example, when sending an additional parameter with a newer implementation, the older implementation can only skip this parameter. ⌋()

[TR_SOMEIP_00218] ⌈ When the receiver of a message adds for example a new parameter to be received, it has to define a default value. This is needed in the case of a sender with an older version of the service sends the message without the additional parameter. ⌋()

[TR_SOMEIP_00219] ⌈ Some changes in the interface specification can be implemented in a non-breaking way:

- Delete Parameters in Functions
  - Need to add default value at receiver first and parameters need to be at end of list
- Remove Exceptions from functions
  - Trivial at server side
  - Client needs to throw "unknown exception", if encountering old exception
- Renaming parameters, functions, and services is possible since the names are not transmitted. The generated code only looks at the IDs and the ordering of parameters, which shall not be changed in migration.

⌋()

[TR_SOMEIP_00220] ⌈ If the struct is configured by the interface specification to have a length field, the following is possible:

- Adding / deleting fields to/from the end of structs

⌋()

[TR_SOMEIP_00221] ⌈ Currently not supported are the following changes:

- Reordering parameters
- Replace supertype by subtype (as in object oriented programming languages with inheritance)

⌋()

6.6 Transporting CAN and FlexRay Frames

[TR_SOMEIP_00222] ⌈ SOME/IP should allows to tunnel CAN and FlexRay frames as well. However, the Message ID space needs to be coordinated between both use cases. ⌋()

[TR_SOMEIP_00223] ⌈ The full SOME/IP Header shall be used for transporting/tunneling CAN/FlexRay. ⌋()

[TR_SOMEIP_00224] ⌈ The AUTOSAR Socket-Adapter uses the Message ID and Length to construct the needed internal PDUs but does not look at other fields. Therefore, the CAN ID (11 or 29 bits) and/or the FlexRay ID (6+6+11 bits) have to be encoded into the Message ID field. The CAN ID shall be aligned to the least significant bit of the Message ID. An 11 bit CAN identifier would be therefore transported in the bit position 21 to 31. Refer also to [TR_SOMEIP_00232] ⌋()

[TR_SOMEIP_00225] ⌈ Especially with the use of 29 Bit CAN-IDs or FlexRay-IDs a lot of the Message ID space is used. In this case it is recommended to bind SOME/IP and CAN/FlexRay transports to different transport protocol ports, so that different ID spaces for the Message IDs exist. ⌋()

[TR_SOMEIP_00226] ⌈ Keep in mind that when transporting a CAN frame of 8 Byte over Ethernet an overhead of up to 100 Bytes might be needed in the near future (using IPv6 and/or security mechanisms). So it is recommended to use larger RPC calls as shown in the first part of the document instead of small CAN like communication. ⌋()

[TR_SOMEIP_00227] ⌈ Client ID and Session ID shall be set to 0x0000. ⌋()

[TR_SOMEIP_00228] ⌈ Depending on the direction, the Message Type shall be set to 0x02 (service provider sends) or 0x00 (service provider receives). The Return Code shall be always set to 0x00.

⌋()

[TR_SOMEIP_00229] ⌈ At least for 11 Bit CAN-IDs the layout of the Message ID shall be followed [TR_SOMEIP_00038] and [TR_SOMEIP_00040]. This means that the 16th bit from the left shall be set to 0 or 1 according to the Message ID (0x00 or 0x02). ⌋*()*

[TR_SOMEIP_00230] ⌈ Protocol Version shall be set according to [TR_SOMEIP_00052]. ⌋*()*

[TR_SOMEIP_00231] ⌈ Interface Version shall be set according to interface specifications. ⌋*()*

[TR_SOMEIP_00232] ⌈ If SOME/IP is used for transporting CAN messages with 11 Bits of CAN-ID, the following layout of the Message ID may is recommended (example):

- Service ID shall be set to a value defined by the system department, e.g. 0x1234
- Event ID is split into 4 Bits specifying the CAN bus, and 11 Bits for the CAN-ID.

This is just an example and the actual layout shall be specified by the System Department. ⌋*()*

6.6.1 AUTOSAR specific

[TR_SOMEIP_00233] ⌈ Some AUTOSAR-implementations currently do not allow for sending more than one CAN or FlexRay frame inside an IP packet. All AUTOSAR implementations shall allow receiving more than one CAN or FlexRay frame inside an IP packet by use of the length field. ⌋*()*

6.7 SOME/IP Service Discovery (SOME/IP-SD)

6.7.1 General

[TR_SOMEIP_00234] ⌈ Service Discovery is used to locate service instances and to detect if service instances are running as well as implementing the Publish/Subscribe handling. ⌋*()*

[TR_SOMEIP_00235] ⌈ Inside the vehicular network service instance locations are commonly known; therefore, the state of the service instance is of primary concern. The location of the service (i.e. IP-Address, transport protocol, and port number) are of secondary concern. ⌋*()*

6.7.1.1 Terms and Definitions

[TR_SOMEIP_00236] ⌈ The Server-Service-Instance-Entry shall include one Interface Identifier of the communication interface (e.g. Ethernet interface or virtual interface based on Ethernet VLANs) the service is offered on. ⌋*()*

[TR_SOMEIP_00237] ⌈ Client-Service-Instance-Entry shall include one Interface Identifier of the communication interface the service is configured to be accessed with. ⌋*()*

[TR_SOMEIP_00238] ⌈ Multiple Server-Service-Instance-Entry entries shall be used, if a service instance needs to be offered on multiple interfaces. ⌋*()*

[TR_SOMEIP_00239] ⌈ Multiple Client-Service-Instance-Entry entries shall be used, if a service instance needs to be configured to be accessed using multiple different interfaces. ⌋*()*

6.7.2 SOME/IP-SD ECU-internal Interface

[TR_SOMEIP_00240] ⌈ Service status shall be defined as up or down as well as required and released:

- A service status of up shall mean that a service instance is available; thus, it is accessible using the communication method specified and is able to fulfil its specified function.
- A service status of down shall mean the opposite of the service status up.
- A service status of required shall mean that service instance is needed by another software component in the system to function.
- A service status of released shall mean the opposite of the service status required.
- The combination of service status up/down with required/released shall be supported. Four different valid combinations shall exist (up+required, up+released, down+required, down+released).

⌋*()*

[TR_SOMEIP_00241] ⌈ The Service Discovery Interface shall inform local software components about the status of remote services (up/down). ⌋*()*

[TR_SOMEIP_00242] ⌈ The Service Discovery Interface shall offer the option to local software component to require or release a remote service instance. ⌋*()*

[TR_SOMEIP_00243] ⌈ The Service Discovery Interface shall inform local software components of the require/release status of local services. ⌋*()*

[TR_SOMEIP_00244] ⌈ The Service Discovery Interface shall offer the option to local software component to set a local service status (up/down). ⌋*()*

[TR_SOMEIP_00245] ⌈ Eventgroup status shall be defined in the same way the service status is defined. ⌋*()*

[TR_SOMEIP_00246] ⌈ Service Discovery shall be used to turn on/off the events and notification events of a given eventgroup. Only if another ECU requires an eventgroup

the events and notification events of this eventgroup are sent. (See Subscribe Event Group). ⌋()

[TR_SOMEIP_00247] ⌈ The Service Discovery shall be informed of link-up and link-down events of logical, virtual, and physical communication interfaces that the Service Discovery is bound to. ⌋()

6.7.3 SOME/IP-SD Message Format

6.7.3.1 General Requirements

[TR_SOMEIP_00248] ⌈ Service Discovery messages shall be supported over UDP. ⌋()

[TR_SOMEIP_00250] ⌈ Service Discovery Messages shall start with a SOME/IP header as depicted Figure 6.11:

- Service Discovery messages shall use the Service-ID (16 Bits) of 0xFFFF.
- Service Discovery messages shall use the Method-ID (16 Bits) of 0x8100.
- Service Discovery messages shall use a uint32 length field as specified by SOME/IP. That means that the length is measured in bytes and starts with the first byte after the length field and ends with the last byte of the SOME/IP-SD message.
- Service Discovery messages shall have a Client-ID (16 Bits) and handle it based on SOME/IP rules.
- The Client ID shall be set to 0, since there exists only a single SOME/IP-SD instance.
- If a Client ID Prefix is configured, it shall also apply to SOME/IP-SD.
- Service Discovery messages shall have a Session-ID (16 Bits) and handle it based on SOME/IP requirements.
- The Session-ID (SOME/IP header) shall be incremented for every SOME/IP-SD message sent.
- The Session-ID (SOME/IP header) shall start with 1 and be 1 even after wrapping.
- The Session-ID (SOME/IP header) shall not be set to 0.
- SOME/IP-SD Session ID handling is done per "communication relation", i.e. broadcast as well as unicast per peer (see [TR_SOMEIP_00255]).
- Service Discovery messages shall have a Protocol-Version (8 Bits) of 0x01.
- Service Discovery messages shall have a Interface-Version (8 Bits) of 0x01.
- Service Discovery messages shall have a Message Type (8 bits) of 0x02 (Notification).

- Service Discovery messages shall have a Return Code (8 bits) of 0x00 (E_OK).

⌋()

[TR_SOMEIP_00251] ⌈

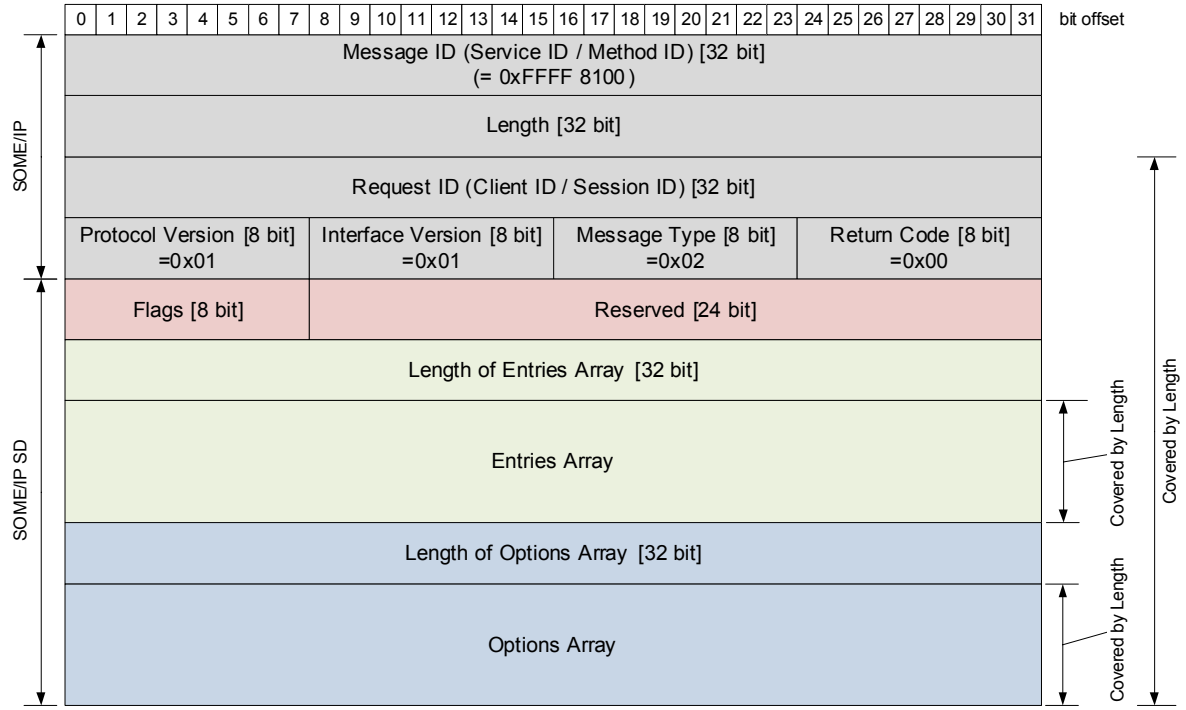

Figure 6.11: SOME/IP-SD Header Format

⌋()

6.7.3.2 SOME/IP-SD Header

[TR_SOMEIP_00252] ⌈ After the SOME/IP header the SOME/IP-SD Header shall follow as depicted in Figure 6.11. ⌋()

[TR_SOMEIP_00253] ⌈ The SOME/IP-SD Header shall start out with an 8 Bit field called Flags. ⌋()

[TR_SOMEIP_00254] ⌈ The first flag of the SOME/IP-SD Flags field (highest order bit) shall be called Reboot Flag. ⌋()

[TR_SOMEIP_00255] ⌈ The Reboot Flag of the SOME/IP-SD Header shall be set to one for all messages after reboot until the Session-ID in the SOME/IP-Header wraps around and thus starts with 1 again. After this wrap around the Reboot Flag is set to 0. ⌋()

[TR_SOMEIP_00256] ⌈ The information for the reboot flag and the Session ID shall be kept for multicast and unicast separately as well as for every sender-receiver-relation (i.e. source address and destination address). ⌋()

Note:
This means you have to store a counter for Multicast sending and one counter per Unicast peer as wellcas two counters (1x Multicast, 1x Unicast) per SOME/IP-SD peer for receiving.

[TR_SOMEIP_00257] ⌈ SOME/IP-SD implementations shall be able to reliably detect the reboots of their peer based on these informations. ⌋*()*

[TR_SOMEIP_00258] ⌈ The detection of a reboot shall be done as follows (with new the values of the current packet from the communication partner and old the last value received before):

if old.reboot==0 and new.reboot==1 then Reboot detected
if old.reboot==1 and new.reboot==1 and old.session_id>=new.session_id then Reboot detected

The following is not enough since we do not have reliable communication:
if new.reboot==1 and old.session_id>=new.session_id then Reboot detected
⌋*()*

[TR_SOMEIP_00525] ⌈ When the system detects the reboot of a peer, it shall update its state accordingly. Services and Subscriptions shall be expired if they are not updated again. ⌋*()*

[TR_SOMEIP_00526] ⌈ When the system detects the reboot of a peer, it shall reset the state of the TCP connections to this peer. The client shall reestablish the TCP as required by the Publish/Subscribe process later. ⌋*()*

[TR_SOMEIP_00259] ⌈ The second flag of the SOME/IP-SD Flags (second highest order bit) shall be called Unicast Flag and shall be set to 1, if SD message reception via unicast is supported. ⌋*()*

[TR_SOMEIP_00540] ⌈ The Unicast Flag of the SOME/IP-SD Header shall be set to Unicast (that means 1) for all SD Messages since this means that receiving using unicast is supported. ⌋*()*

[TR_SOMEIP_00261] ⌈ After the Flags the SOME/IP-SD Header shall have a field of 24 bits called Reserved that is set to 0 until further notice. ⌋*()*

[TR_SOMEIP_00262] ⌈ After the SOME/IP-SD Header the Entries Array shall follow. ⌋*()*

[TR_SOMEIP_00263] ⌈ The entries shall be processed exactly in the order they arrive. ⌋*()*

[TR_SOMEIP_00578] ⌈ [TR_SOMEIP_00263] shall not lead to closing and reopening a socket because of a single Service Discovery message. The socket shall stay open in this case. ⌋*()*

[TR_SOMEIP_00264] ⌈ After the Entries Array in the SOME/IP-SD Header an Option Array shall follow. ⌋*()*

[TR_SOMEIP_00265] ⌈ The Entries Array and the Options Array of the SOME/IP-SD message shall start with a length field as uint32 that counts the number of bytes of the following data; i.e. the entries or the options. ⌋*()*

6.7.3.3 Entry Format

[TR_SOMEIP_00266] ⌈ The service discovery shall work on different entries that shall be carried in the service discovery messages. The entries are used to synchronize the state of services instances and the Publish/Subscribe handling. ⌋*()*

[TR_SOMEIP_00267] ⌈ Two types of entries exist: A Service Entry Type for Services and an Eventgroup Entry Type for Eventgroups, which are used for different entries each. ⌋*()*

[TR_SOMEIP_00268] ⌈ A Service Entry Type shall be 16 Bytes of size and include the following fields in this order as shown in Figure 6.12:

- Type Field [uint8]: encodes FindService (0x00) and OfferService (0x01).
- Index First Option Run [uint8]: Index of this runs first option in the option array.
- Index Second Option Run [uint8]: Index of this runs second option in the option array.
- Number of Options 1 [uint4]: Describes the number of options the first option run uses.
- Number of Options 2 [uint4]: Describes the number of options the second option run uses.
- Service-ID [uint16]: Describes the Service ID of the Service or Service-Instance this entry is concerned with.
- Instance ID [uint16]: Describes the Service Instance ID of the Service Instance this entry is concerned with or is set to 0xFFFF if all service instances of a service are meant.
- Major Version [uint8]: Encodes the major version of the service (instance).
- TTL [uint24]: Describes the lifetime of the entry in seconds.
- Minor Version [uint32]: Encodes the minor version of the service.

⌋*()*

[TR_SOMEIP_00269] ⌈

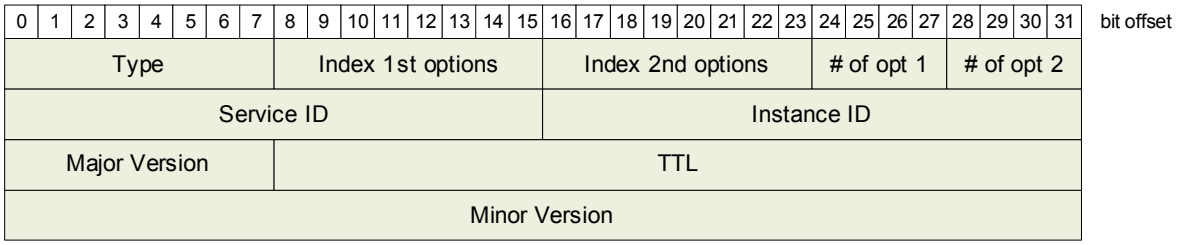

Figure 6.12: SOME/IP-SD Service Entry Type

⌋()

[TR_SOMEIP_00270] ⌈ An Eventgroup Entry shall be 16 Bytes of size and include the following fields in this order as shown in Figure 6.13:

- Type Field [uint8]: encodes Subscribe (0x06), and SubscribeAck (0x07).
- Index First Option Run [uint8]: Index of this runs first option in the option array.
- Index Second Option Run [uint8]: Index of this runs second option in the option array.
- Number of Options 1 [uint4]: Describes the number of options the first option run uses.
- Number of Options 2 [uint4]: Describes the number of options the second option run uses.
- Service-ID [uint16]: Describes the Service ID of the Service or Service Instance this entry is concerned with.
- Instance ID [uint16]: Describes the Service Instance ID of the Service Instance this entry is concerned with or is set to 0xFFFF if all service instances of a service are meant.
- Major Version [uint8]: Encodes the major version of the service instance this eventgroup is part of.
- TTL [uint24]: Descibes the lifetime of the entry in seconds.
- Reserved [uint12]: Shall be set to 0x000, if not specified otherwise (see [TR_SOMEIP_00391] and [TR_SOMEIP_00394]).
- Counter [uint4]: Is used to differentiate identical Subscribe Eventgroups. Set to 0x0 if not used.
- Eventgroup ID [uint16]: Transports the ID of an Eventgroup.

⌋()

[TR_SOMEIP_00271] ⌈

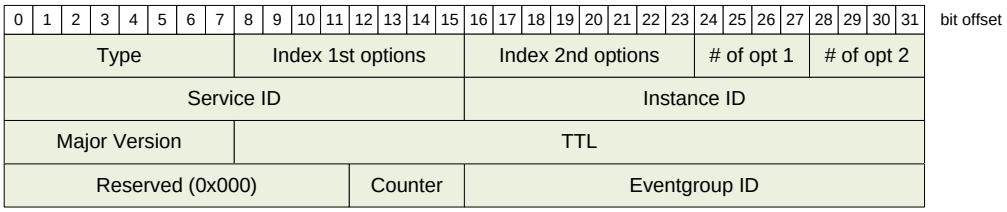

Figure 6.13: SOME/IP-SD Eventgroup Entry Type

⌋()

6.7.3.4 Options Format

[TR_SOMEIP_00272] ⌈ Options are used to transport additional information to the entries. This includes for instance the information how a service instance is reachable (IP-Address, Transport Protocol, Port Number). ⌋()

[TR_SOMEIP_00273] ⌈ In order to identify the option type every option shall start with:

- Length [uint16]: Specifies the length of the option in Bytes.
- Type [uint8]: Specifying the type of the option.

⌋()

[TR_SOMEIP_00274] ⌈ The length field shall cover all bytes of the option except the length field and type field. ⌋()

6.7.3.4.1 Configuration Option

[TR_SOMEIP_00275] ⌈ The configuration option shall be used to transport arbitrary configuration strings. This allows to encode additional information like the name of a service or its configuration. ⌋()

[TR_SOMEIP_00276] ⌈ The format of the Configuration Option shall be as follows:

- Length [uint16]: Shall be set to the total number of bytes occupied by the configuration option, excluding the 16 bit length field and the 8 bit type flag.
- Type [uint8]: Shall be set to 0x01.
- Reserved [uint8]: Shall be set to 0x00.
- ConfigurationString [dyn length]: Shall carry the configuration string.

⌋()

[TR_SOMEIP_00277] ⌈ The Configuration Option shall specify a set of name-value-pairs based on the DNS TXT and DNS-SD format. ⌋()

[TR_SOMEIP_00278] ⌈ The format of the configuration string shall start with a single byte length field that describes the number of bytes following this length field. ⌋()

[TR_SOMEIP_00279] ⌈ After each character sequence another length field and a following character sequence are expected until a length field set to 0x00. ⌋()

[TR_SOMEIP_00280] ⌈ After a length field set to 0x00 no characters follow. ⌋()

[TR_SOMEIP_00281] ⌈ A character sequence shall encode a key and an optionally a value. ⌋()

[TR_SOMEIP_00282] ⌈ The character sequences shall contain an equal character ("=", 0x03D) to devide key and value. ⌋()

[TR_SOMEIP_00283] ⌈ The key shall not include an equal character and shall be at least one non-whitespace character. The characters of "Key" shall be printable US-ASCII values (0x20-0x7E), excluding '=' (0x3D). ⌋()

[TR_SOMEIP_00284] ⌈ The "=" shall not be the first character of the sequence. ⌋()

[TR_SOMEIP_00285] ⌈ For a character sequence without an '=' that key shall be interpreted as present. ⌋()

[TR_SOMEIP_00286] ⌈ For a character sequence ending on an '=' that key shall be interpreted as present with empty value. ⌋()

[TR_SOMEIP_00287] ⌈ Multiple entries with the same key in a single Configuration Option shall be supported. ⌋()

[TR_SOMEIP_00288] ⌈ The configuration option shall be also used to encode hostname, servicename, and instancename (if needed). ⌋()

[TR_SOMEIP_00289] ⌈ Figure 6.14 shows the format of the Configuration Option and Figure 6.15 an example for the Configuration Option. ⌋()

[TR_SOMEIP_00290] ⌈

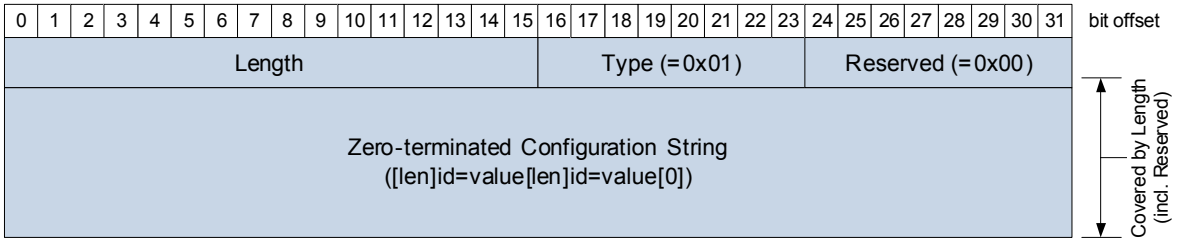

Figure 6.14: SOME/IP-SD Configuration Option

⌋()

[TR_SOMEIP_00291] ⌈

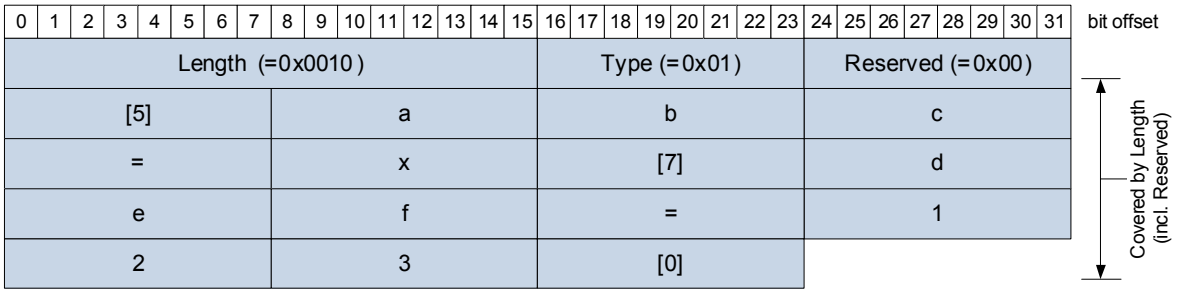

Figure 6.15: SOME/IP-SD Configuration Option Example

⌋()

6.7.3.4.2 Load Balancing Option (informational)

[TR_SOMEIP_00541] ⌈ This option shall be used to prioritize different instances of a service, so that a client chooses the service instance based on these settings. This option will be attached to Offer Service entries. ⌋()

[TR_SOMEIP_00542] ⌈ The Load Balancing Option shall carry a Priority and Weight like the DNS-SRV records, which canshall be used to load balancing different service instances. ⌋()

[TR_SOMEIP_00543] ⌈ This means when looking for all service instances of a service (Service Instance set to 0xFFFF), the client shall choose the service instance with highest priority. When having more than one service instance with highest priority (lowest value in Priority field) the service instance shall be chosen randomly based on the weights of the service instances. ⌋()

[TR_SOMEIP_00544] ⌈ The Format of the Load Balancing Option shall be as follows:

- Length [uint16]: Shall be set to 0x0005.
- Type [uint8]: Shall be set to 0x02.
- Reserved [uint8]: Shall be set to 0x00.
- Priority [uint16]: Carries the Priority of this instance. Lower value means higher priority.
- Weight [uint16]: Carries the Weight of this instance. Large value means higher probability to be chosen.

⌋()

[TR_SOMEIP_00584] ⌈ The format of the Load Balancing Option shall follow this figure:

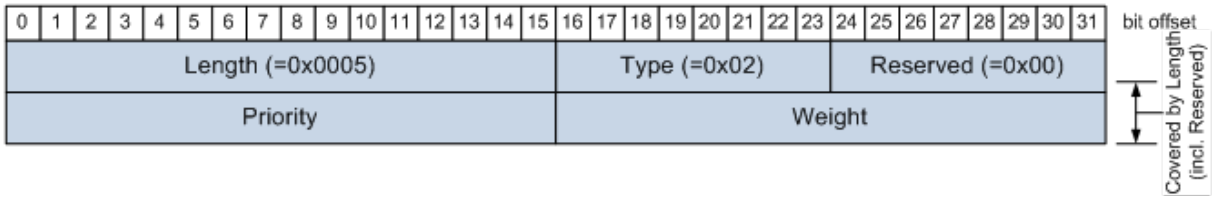

Figure 6.16: SOME/IP-SD Load Balancing Option

⌋()

6.7.3.4.3 IPv4 Endpoint Option

[TR_SOMEIP_00304] ⌈ The IPv4 Endpoint Option shall be used by a SOME/IP-SD instance to signal the relevant endpoint(s). Endpoints include the local IP address, the transport layer protocol (e.g. UDP or TCP), and the port number of the sender.

These ports shall be used for the events and notification events as well. When using UDP the server uses the announced port as ssource port. With TCP the client needs to open a connection to this port before subscription because this is the TCP connection the server uses for sending events and notification events. ⌋()

[TR_SOMEIP_00305] ⌈ The IPv4 Endpoint Option shall use the Type 0x24. ⌋()

[TR_SOMEIP_00306] ⌈ The IPv4 Endpoint Option shall specify the IPv4-Address, the transport layer protocol (ISO/OSI layer 4) used, and its Port Number. ⌋()

[TR_SOMEIP_00307] ⌈ The Format of the IPv4 Endpoint Option shall be as follows:

- Length [uint16]: Shall be set to 0x0009.
- Type [uint8]: Shall be set to 0x24.
- Reserved [uint8]: Shall be set to 0x00.
- IPv4-Address [uint32]: Shall transport the unicast IP-Address of SOME/IP-SD as four Bytes.
- Reserved [uint8]: Shall be set to 0x00.
- Transport Protocol (L4-Proto) [uint8]: Shall be set to the transport layer protocol (ISO/OSI layer 4) based on the IANA/IETF types (0x06: TCP, 0x11: UDP).
- Transport Protocol Port Number (L4-Port) [uint16]: Shall be set to the transport layer port of SOME/IP-SD (e.g. 30490).

⌋()

[TR_SOMEIP_00308] ⌈ Figure 6.17 shows the format of the IPv4 Endpoint Option. ⌋()

[TR_SOMEIP_00309] ⌈

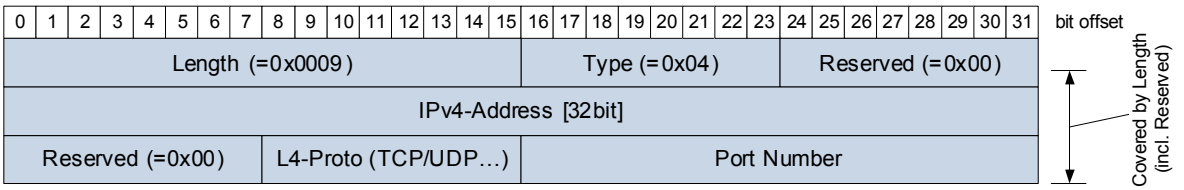

Figure 6.17: SOME/IP-SD IPv4 Endpoint Option

⌋()

[TR_SOMEIP_00310] ⌈ The server shall use the IPv4 Endpoint Option with Offer Service entries to signal the endpoints it serves the service on. That is upto one UDP endpoint and upto one TCP endpoint. ⌋()

[TR_SOMEIP_00311] ⌈ The endpoints the server referenced with an Offer Service entry shall also be used as source of events. That is source IP address and source port numbers for the transport protocols in the endpoint option. ⌋()

[TR_SOMEIP_00312] ⌈ The client shall use the IPv4 Endpoint Options with Subscribe Eventgroup entries to signal its IP address and its UDP and/or TCP port numbers, on which it is ready to receive the events. ⌋*()*

6.7.3.4.4 IPv6 Endpoint Option

[TR_SOMEIP_00313] ⌈ The IPv6 Endpoint Option shall be used by a SOME/IP-SD instance to signal the relevant endpoint(s). Endpoints include the local IP address, the transport layer protocol (e.g. UDP or TCP), and the port number of the sender.

These ports shall be used for the events and notification events as well. When using UDP the server uses the announced port as source port. With TCP the client needs to open a connection to this port before subscription because this is the TCP connection the server uses for sending events and notification events. ⌋*()*

[TR_SOMEIP_00314] ⌈ The IPv6 Endpoint Option shall use the Type 0x24. ⌋*()*

[TR_SOMEIP_00315] ⌈ The IPv6 Endpoint Option shall specify the IPv6-Address, the Layer 4 Protocol used, and the Layer 4 Port Number. ⌋*()*

[TR_SOMEIP_00316] ⌈ The Format of the IPv6 Endpoint Option shall be as follows:

- Length [uint16]: Shall be set to 0x0015.
- Type [uint8]: Shall be set to 0x24.
- Reserved [uint8]: Shall be set to 0x00.
- IPv6-Address [uint128]: Shall transport the unicast IP-Address of SOME/IP-SD as 16 Bytes
- Reserved [uint8]: Shall be set to 0x00.
- Transport Protocol (L4-Proto) [uint8]: Shall be set to the transport layer protocol (ISO/OSI layer 4) based on the IANA/IETF types (0x06: TCP, 0x11: UDP).
- Transport Protocol Port Number (L4-Port) [uint16]: Shall be set to the transport layer port of SOME/IP-SD (e.g. 30490).

⌋*()*

[TR_SOMEIP_00317] ⌈ Figure 6.18 shows the format of the IPv6 Endpoint Option. ⌋*()*

[TR_SOMEIP_00318] ⌈

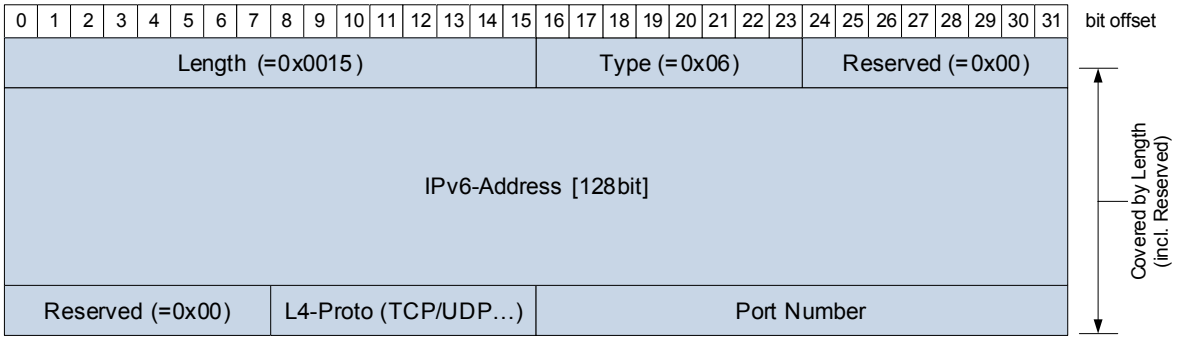

Figure 6.18: SOME/IP-SD IPv6 Endpoint Option

⌋()

[TR_SOMEIP_00319] ⌈ The server shall use the IPv6 Endpoint Option with Offer Service entries to signal the endpoints the services is available on. That is upto one UDP endpoint and upto one TCP endpoint. ⌋()

[TR_SOMEIP_00320] ⌈ The endpoints the server referenced with an Offer Service entry shall also be used as source of events. That is source IP address and source port numbers for the transport protocols in the endpoint option. ⌋()

[TR_SOMEIP_00321] ⌈ The client shall use the IPv6 Endpoint Option with Subscribe Eventgroup entries to signal the IP address and the UDP and/or TCP port numbers, on which it is ready to receive the events. ⌋()

6.7.3.4.5 IPv4 Multicast Option

[TR_SOMEIP_00322] ⌈ The IPv4 Multicast Option is used by the server to announce the IPv4 multicast address, the transport layer protocol (ISO/OSI layer 4), and the port number the multicast events and multicast notification events are sent to. As transport layer protocol currently only UDP is supported. ⌋()

[TR_SOMEIP_00323] ⌈ The IPv4 Multicast Option and not the IPv4 Endpoint Option shall be referenced by Subscribe Eventgroup Ack entries. ⌋()

[TR_SOMEIP_00324] ⌈ The IPv4 Multicast Option shall use the Type 0x14. ⌋()

[TR_SOMEIP_00325] ⌈ The IPv4 Multicast Option shall specify the IPv4-Address, the transport layer protocol (ISO/OSI layer 4) used, and its Port Number. ⌋()

[TR_SOMEIP_00326] ⌈ The Format of the IPv4 Endpoint Option shall be as follows:

- Length [uint16]: Shall be set to 0x0009.
- Type [uint8]: Shall be set to 0x14.
- Reserved [uint8]: Shall be set to 0x00.
- IPv4-Address [uint32]: Shall transport the multicast IP-Address as four Bytes.

- Reserved [uint8]: Shall be set to 0x00.
- Transport Protocol (L4-Proto) [uint8]: Shall be set to the transport layer protocol (ISO/OSI layer 4) based on the IANA/IETF types (0x11: UDP).
- Transport Protocol Port Number (L4-Port) [uint16]: Shall be set to the port of the layer 4 protocol.

⌋()

[TR_SOMEIP_00327] ⌈ Figure 6.19 shows the format of the IPv4 Multicast Option. ⌋()

[TR_SOMEIP_00328] ⌈

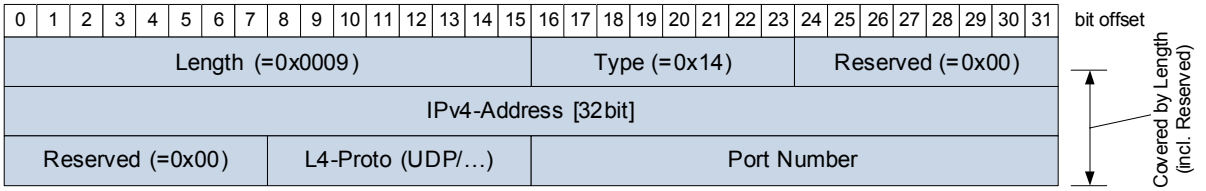

Figure 6.19: SOME/IP-SD IPv4 Multicast Option

⌋()

[TR_SOMEIP_00329] ⌈ The server shall reference the IPv4 Multicast Option, which encodes the IPv4 Multicast Address and Port Number the server will send multicast events and notification events to. ⌋()

6.7.3.4.6 IPv6 Multicast Option

[TR_SOMEIP_00330] ⌈ The IPv6 Multicast Option is used by the server to announce the IPv6 multicast address, the layer 4 protocol, and the port number the multicast events and multicast notifications events are sent to. For the transport layer protocol (ISO/OSI layer 4) currently only UDP is supported. ⌋()

[TR_SOMEIP_00545] ⌈ The IPv6 Multicast Option and not the IPv6 Endpoint Option shall be referenced by Subscribe Eventgroup Ack messages. ⌋()

[TR_SOMEIP_00331] ⌈ The IPv6 Multicast Option shall use the Type 0x16. ⌋()

[TR_SOMEIP_00332] ⌈ The IPv6 Multicast Option shall specify the IPv6-Address, the transport layer protocol (ISO/OSI layer 4) used, and its Port Number. ⌋()

[TR_SOMEIP_00333] ⌈ The Format of the IPv6 Multicast Option shall be as follows:

- Length [uint16]: Shall be set to 0x0015.
- Type [uint8]: Shall be set to 0x16.
- Reserved [uint8]: Shall be set to 0x00.
- IPv6-Address [uint128]: Shall transport the multicast IP-Address as 16 Bytes.

- Reserved [uint8]: Shall be set to 0x00.
- Transport Protocol (L4-Proto) [uint8]: Shall be set to the transport layer protocol (ISO/OSI layer 4) based on the IANA/IETF types (0x11: UDP).
- Transport Protocol Port Number (L4-Port) [uint16]: Shall be set to the port of the layer 4 protocol.

⌋()

[TR_SOMEIP_00334] ⌈ Figure 6.20 shows the format of the IPv6 Multicast Option. ⌋()

[TR_SOMEIP_00335] ⌈

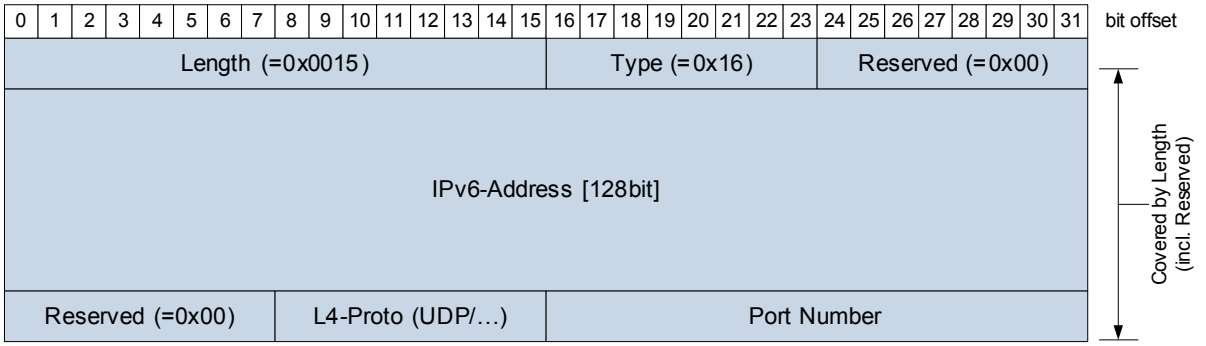

Figure 6.20: SOME/IP-SD IPv6 Multicast Option

⌋()

[TR_SOMEIP_00336] ⌈ The server shall reference the IPv6 Multicast Option, which encodes the IPv6 Multicast Address and Port Number the server will send multicast events and notification events to. ⌋()

6.7.3.4.7 IPv4 SD Endpoint Option

The IPv4 SD Endpoint Option is used to transport the endpoint (i.e. IP-Address and Port) of the senders SD implementation. This is used to identify the SOME/IP-SD Instance even in cases in which the IP-Address and/or Port Number cannot be used.

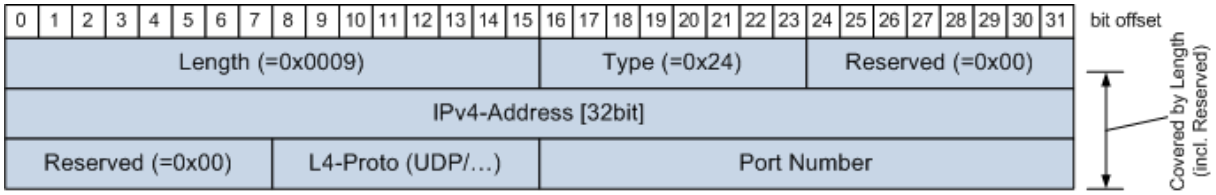

Figure 6.21: SOME/IP-SD IPv4 Endpoint Option

[TR_SOMEIP_00546] ⌈ The IPv4 SD Endpoint Option is used to transport the endpoint (i.e. IP-Address and Port) of the senders SD implementation. ⌋()

This is used to identify the SOME/IP-SD Instance even in cases in which the IP-Address and/or Port Number cannot be used.

[TR_SOMEIP_00547] ⌈ The IPv4 SD Endpoint Option shall be included in any SD message up to 1 time. ⌋*()*

[TR_SOMEIP_00650] ⌈ The IPv4 SD Endpoint Option shall only be included if the SOME/IP-SD message is transported over IPv4. ⌋*()*

[TR_SOMEIP_00651] ⌈ The IPv4 SD Endpoint Option shall be the first option in the options array, if existing. ⌋*()*

[TR_SOMEIP_00652] ⌈ If more than one IPv4 SD Endpoint Option is received, only the first one shall be processed and all further IPv4 SD Endpoint Options shall be ignored. ⌋*()*

[TR_SOMEIP_00548] ⌈ The IPv4 SD Endpoint Option shall be not referenced by any SD Entry. ⌋*()*

[TR_SOMEIP_00549] ⌈ If the IPv4 SD Endpoint Option is included in the SD message, the receiving SD implementation shall use the content of this option instead of the Source IP Address and Source Port. ⌋*()*

Note:
This is important for answering the received SD message (e.g. Offer after Find or Subscribe after Offer or Subscribe Ack after Subscribe) as well as the reboot detection (channel based on SD Endpoint Option and not out addresses).

[TR_SOMEIP_00550] ⌈ The IPv4 SD Endpoint Option shall use the Type 0x24. ⌋*()*

[TR_SOMEIP_00551] ⌈ The IPv4 SD Endpoint Option shall specify the IPv4-Address, the transport layer protocol (ISO/OSI layer 4) used, and a Port Number. ⌋*()*

[TR_SOMEIP_00552] ⌈ The Format of the IPv4 SD Endpoint Option shall be as follows:

- Length [uint16]: Shall be set to 0x0015.
- Type [uint8]: Shall be set to 0x24.
- Reserved [uint8]: Shall be set to 0x00.
- IPv4-Address [uint32]: Shall transport the unicast IP-Address of SOME/IP-SD as four Bytes.
- Reserved [uint8]: Shall be set to 0x00.
- Transport Protocol (L4-Proto) [uint8]: Shall be set to the transport layer protocol of SOME/IP-SD (currently: 0x11 UDP).
- Transport Protocol Port Number (L4-Port) [uint16]: Shall be set to the transport layer port of SOME/IP-SD (currently: 30490).

⌋*()*

6.7.3.4.8 IPv6 SD Endpoint Option

The Ipv6 SD Endpoint Option is used to transport the endpoint (i.e. IP-Address and Port) of the senders SD implementation. This is used to identify the SOME/IP-SD Instance even in cases in which the IP-Address and/or Port Number cannot be used.

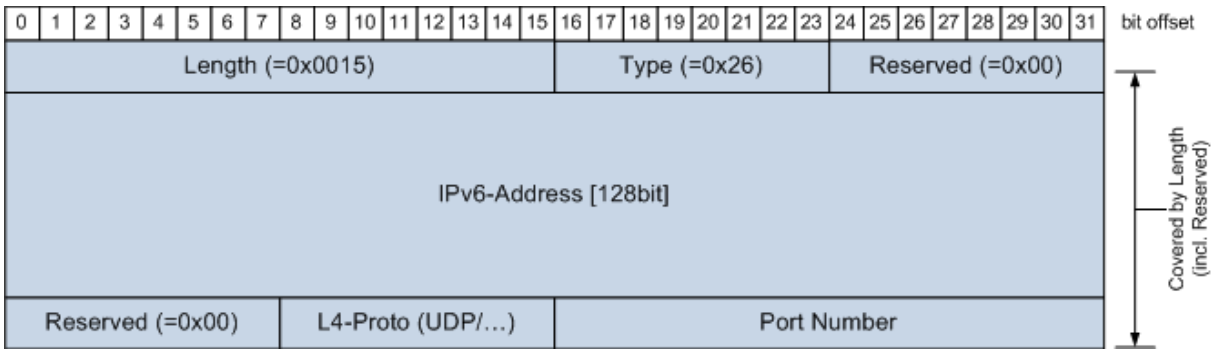

Figure 6.22: SOME/IP-SD IPv6 Endpoint Option

[TR_SOMEIP_00554] ⌈ The IPv6 SD Endpoint Option may be included in any SD message up to 1 time. ⌋*()*

[TR_SOMEIP_00653] ⌈ The IPv6 SD Endpoint Option shall only be included if the SOME/IP-SD message is transported over IPv6. ⌋*()*

[TR_SOMEIP_00654] ⌈ The IPv6 SD Endpoint Option shall be the first option in the options array, if existing. ⌋*()*

[TR_SOMEIP_00655] ⌈ If more than one IPv6 SD Endpoint Option is received, only the first one shall be processed and all further IPv6 SD Endpoint Options shall be ignored ⌋*()*

[TR_SOMEIP_00555] ⌈ The IPv6 SD Endpoint Option shall be not referenced by any SD Entry. ⌋*()*

[TR_SOMEIP_00556] ⌈ If the IPv6 SD Endpoint Option is included in the SD message, the receiving SD implementation shall use the content of this option instead of the Source IP Address and Source Port for answering this SD messages. ⌋*()*

This is important for answering the received SD messages (e.g. Offer after Find or Subscribe after Offer or Subscribe Ack after Subscribe) as well as the reboot detection (channel based on SD Endpoint Option and not out addresses).

[TR_SOMEIP_00557] ⌈ The IPv6 SD Endpoint Option shall use the Type 0x24. ⌋*()*

[TR_SOMEIP_00558] ⌈ The IPv6 SD Endpoint Option shall specify the IPv6-Address, the transport layer protocol (ISO/OSI layer 4) used, and its Port Number. ⌋*()*

[TR_SOMEIP_00559] ⌈ The Format of the IPv6 SD Endpoint Option shall be as follows:

- Length [uint16]: Shall be set to 0x0015.
- Type [uint8]: Shall be set to 0x24.

- Reserved [uint8]: Shall be set to 0x00.
- IPv6-Address [uint128]: Shall transport the unicast IP-Address of SOME/IP-SD as 16 Bytes.
- Reserved [uint8]: Shall be set to 0x00.
- Transport Protocol (L4-Proto) [uint8]: Shall be set to the transport layer protocol of SOME/IP-SD (currently: 0x11 UDP).
- Transport Protocol Port Number (L4-Port) [uint16]: Shall be set to the transport layer port of SOME/IP-SD (currently: 30490).

⌋()

6.7.3.4.9 Security Considerations for SOME/IP-SD Options

[TR_SOMEIP_00656] ⌈ A SOME/IP-SD implementation shall always check that the IP Addresses received in Endpoint options and SD Endpoint options are topological correct (reference IP Addresses in the IP subnet for which SOME/IP-SD is used) and shall ignore IP Addresses that are not topological correct as well as the entries referencing those options. ⌋()

Note:
This means that only Clients and Servers in the same subset are accesible. An example for checking the IP Addresses (Endpoint-IP) for topological correctness is: SOME/IP-SD-IP-Address AND Netmask = Endpoint-IP AND Netmask.

6.7.3.5 Referencing Options from Entries

[TR_SOMEIP_00337] ⌈ Using the following fields of the entries, options are referenced by the entries:

- Index First Option Run: Index into array of options for first option run. Index 0 means first of SOME/IP-SD packet.
- Index Second Option Run: Index into array of options for second option run. Index 0 means first of SOME/IP-SD packet.
- Number of Options 1: Length of first option run. Length 0 means no option in option run.
- Number of Options 2: Length of second option run. Length 0 means no option in option run.

⌋()

[TR_SOMEIP_00339] ⌈ Two different option runs exist: First Option Run and Second Option Run. ⌋()

Rationale for the support of two option runs: Two different types of options are expected: options common between multiple SOME/IP-SD entries and options different for each SOME/IP-SD entry. Supporting to different options runs is the most efficient way to support these two types of options, while keeping the wire format highly efficient.

[TR_SOMEIP_00341] ⌈ Each option run shall reference the first option and the number of options for this run. ⌋*()*

[TR_SOMEIP_00342] ⌈ If the number of options is set to zero, the option run is considered empty. ⌋*()*

[TR_SOMEIP_00343] ⌈ For empty runs the Index (i.e. Index First Option Run and/or Index Second Option Run) shall be set to zero. ⌋*()*

[TR_SOMEIP_00344] ⌈ Implementations shall accept and process incoming SD messages with option run length set to zero and option index not set to zero. ⌋*()*

[TR_SOMEIP_00560] ⌈ Implementations shall minimize the size of the SD messages by not duplicating Options without need. ⌋*()*

6.7.3.6 Handling missing, redundant, and conflicting Options

[TR_SOMEIP_00561] ⌈ If an entry references an unknown option, this option shall be ignored. ⌋*()*

[TR_SOMEIP_00562] ⌈ If an entry references an redundant option (option that is not needed by this specific entry), this option shall be ignored. ⌋*()*

Example:
A Service needs only a TCP Endpoint but Endpoint Options for UDP and TCP are referenced by the Offer Service entry. UDP endpoint shall be ignored.

[TR_SOMEIP_00563] ⌈ If an entry references two or more options that are in conflict, this entry shall be ignored or answered negatively. ⌋*()*

Example:
An Offer Service entry referencing two Endpoint Options stating two different UDP Ports shall be ignored.

Example:
A Subscribe Eventgroup entry referencing two Endpoint Options stating two different UDP Ports shall be answered with a Subscribe Eventgroup Nack.

[TR_SOMEIP_00564] ⌈ When two different Configuration Options are referenced by an entry, the configuration sets shall be merged. ⌋*()*

[TR_SOMEIP_00565] ⌈ If the two Configuration Options have conflicting items (same name), all items shall be handled. There shall be no attempt been made to merge duplicate items. ⌋*()*

6.7.3.7 Example

[TR_SOMEIP_00345] ⌈ Figure 6.23 shows an example SOME/IP-SD PDU. ⌋()

[TR_SOMEIP_00346] ⌈

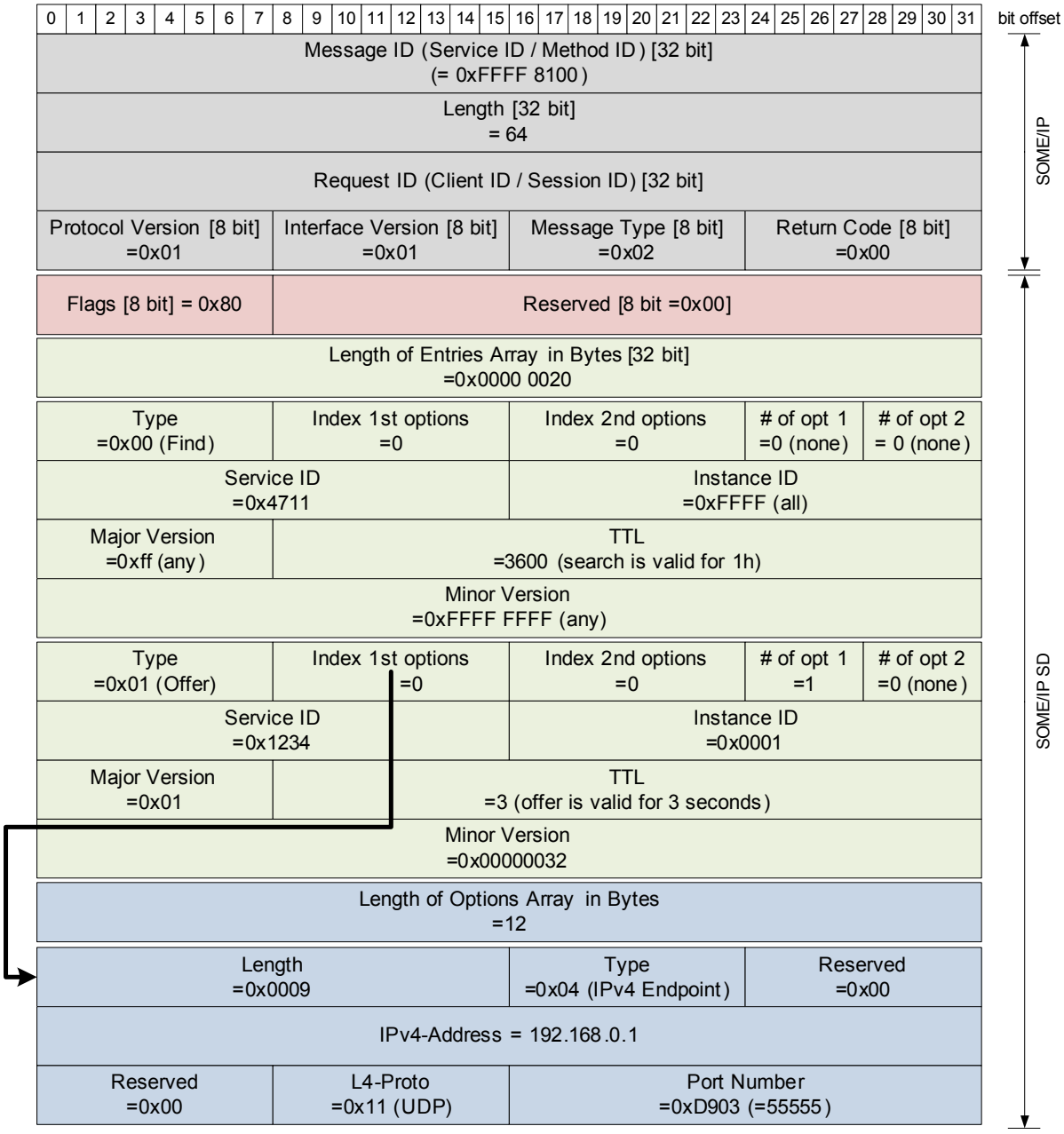

Figure 6.23: SOME/IP-SD Example PDU

⌋()

6.7.4 Service Discovery Messages

[TR_SOMEIP_00347] ⌈ Using the previously specified header format, different entries and messages consisting of one or more entries can be built. The specific entries and their header layouts are explained in the following sections. ⌋*()*

[TR_SOMEIP_00348] ⌈ For all entries the following shall be true:

- Index First Option Run, Index Second Option Run, Number of Options 1, and Number of Options 2 shall be set according to the chained options.

⌋*()*

6.7.4.1 Service Entries

[TR_SOMEIP_00349] ⌈ Entries concerned with services shall be based on the Service Entry Type Format as specified in [TR_SOMEIP_00268]. ⌋*()*

6.7.4.1.1 Find Service Entry

[TR_SOMEIP_00350] ⌈ The Find Service entry type shall be used for finding service instances and shall only be sent if the current state of a service is unknow (no current Service Offer was received and is still valid). ⌋*()*

[TR_SOMEIP_00351] ⌈ Find Service entries shall set the entry fields in the following way:

- Type shall be set to 0x00 (FindService).
- Service ID shall be set to the Service ID of the service that shall be found.
- Instance ID shall be set to 0xFFFF, if all service instances shall be returned. It shall be set to the Instance ID of a specific service instance, if just a single service instance shall be returned.
- Major Version shall be set to 0xFF, that means that services with any version shall be returned. If set to value different than 0xFF, services with this specific major version shall be returned only.
- Minor Version shall be set to 0xFFFF FFFF, that means that services with any version shall be returned. If set to a value different to 0xFFFF FFFF, services with this specific minor version shall be returned only.
- TTL shall be set to the lifetime of the Find Service entry. After this lifetime the Find Service entry shall be considered not existing.
- If set to 0xFFFFFF, the Find Service entry shall be considered valid until the next reboot.

- TTL shall not be set to 0x000000 since this is considered to be the Stop entry for this entry.

⌋()

6.7.4.1.2 Offer Service Entry

[TR_SOMEIP_00355] ⌈ The Offer Service entry type shall be used to offer a service to other communication partners. ⌋()

[TR_SOMEIP_00356] ⌈ Offer Service entries shall set the entry fields in the following way:

- Type shall be set to 0x01 (OfferService).
- Service ID shall be set to the Service ID of the service instance offered.
- Instance ID shall be set to the Instance ID of the service instance that is offered.
- Major Version shall be set to the Major Version of the service instance that is offered.
- Minor Version shall be set to the Minor Version of the service instance that is offered.
- TTL shall be set to the lifetime of the service instance. After this lifetime the service instance shall considered not been offered.
- If set to 0xFFFFFF, the Offer Service entry shall be considered valid until the next reboot.
- TTL shall not be set to 0x000000 since this is considered to be the Stop entry for this entry.

⌋()

[TR_SOMEIP_00357] ⌈ Offer Service entries shall always reference at least an IPv4 or IPv6 Endpoint Option to signal how the service is reachable. ⌋()

[TR_SOMEIP_00358] ⌈ For each Transport Layer Protocol needed for the service (i.e. UDP and/or TCP) an IPv4 Endpoint option shall be added if IPv4 is supported. ⌋()

[TR_SOMEIP_00359] ⌈ For each Transport Layer Protocol needed for the service (i.e. UDP and/or TCP) an IPv6 Endpoint option shall be added if IPv6 is supported. ⌋()

[TR_SOMEIP_00360] ⌈ The IP addresses and port numbers of the Endpoint Options shall also be used for transporting events and notification events: ⌋()

[TR_SOMEIP_00361] ⌈ In the case of UDP this information is used for the source address and the source port of the events and notification events. ⌋()

[TR_SOMEIP_00362] ⌈ In the case of TCP this is the IP address and port the client needs to open a TCP connection to in order to receive events using TCP. ⌋()

6.7.4.1.3 Stop Offer Service Entry

[TR_SOMEIP_00363] ⌈ The Stop Offer Service entry type shall be used to stop offering service instances. ⌋*()*

[TR_SOMEIP_00364] ⌈ Stop Offer Service entries shall set the entry fields exactly like the Offer Service entry they are stopping, except:

- TTL shall be set to 0x000000.

⌋*()*

6.7.4.2 Eventgroup Entry

[TR_SOMEIP_00374] ⌈ Entries concerned with services follow the Eventgroup Entry Type Format as specified in [TR_SOMEIP_00270]. ⌋*()*

6.7.4.2.1 Subscribe Eventgroup Entry

[TR_SOMEIP_00385] ⌈ The Subscribe Eventgroup entry type shall be used to subscribe to an eventgroup. ⌋*()*

[TR_SOMEIP_00386] ⌈ Subscribe Eventgroup entries shall set the entry fields in the following way:

- Type shall be set to 0x06 (SubscribeEventgroup).
- Service ID shall be set to the Service ID of the service instance that includes the eventgroup subscribed to.
- Instance ID shall be set to the Instance ID of the service instance that includes the eventgroup subscribed to.
- Major Version shall be set to the Major Version of the service instance of the eventgroup subscribed to.
- Eventgroup ID shall be set to the Eventgroup ID of the eventgroup subscribed to.
- Major Version shall be set to the Major Version of the service instance that includes the eventgroup subscribed to.
- Counter shall be used to differentiate between parallel subscribes to the same eventgroup of the same service (only difference in endpoint). If not used, set to 0x0.
- Reserved shall be set to 0x000 until further notice.
- TTL shall be set to the lifetime of the eventgroup. After this lifetime the eventgroup shall considered not been subscribed to.

- If set to 0xFFFFFF, the Subscribe Eventgroup entry shall be considered valid until the next reboot.
- TTL shall not be set to 0x000000 since this is considered to be the Stop entry for this entry.

⌋()

[TR_SOMEIP_00387] ⌈ Subscribe Eventgroup entries shall reference one or two IPv4 and/or one or two IPv6 Endpoint Options (one for UDP, one for TCP). ⌋()

6.7.4.2.2 Stop Subscribe Eventgroup Entry

[TR_SOMEIP_00388] ⌈ The Stop Subscribe Eventgroup entry type shall be used to stop subscribing to eventgroups. ⌋()

[TR_SOMEIP_00389] ⌈ Stop Subscribe Eventgroup entries shall set the entry fields exactly like the Subscribe Eventgroup entry they are stopping, except:

- TTL shall be set to 0x000000.

⌋()

[TR_SOMEIP_00574] ⌈ A Stop Subscribe Eventgroup Entry shall reference the same options the Subscribe Eventgroup Entry referenced. This includes but is not limited to Endpoint and Configuration options. ⌋()

6.7.4.2.3 Subscribe Eventgroup Acknowledgement (Subscribe Eventgroup Ack) Entry

[TR_SOMEIP_00390] ⌈ The Subscribe Eventgroup Acknowledgment entry type shall be used to indicate that Subscribe Eventgroup entry was accepted. ⌋()

[TR_SOMEIP_00391] ⌈ Subscribe Eventgroup Acknowledgment entries shall set the entry fields in the following way:

- Type shall be set to 0x07 (SubscribeEventgroupAck).
- Service ID, Instance ID, Major Version, Eventgroup ID, TTL, Counter, and Reserved shall be the same value as in the Subscribe that is being answered.

⌋()

[TR_SOMEIP_00392] ⌈ Subscribe Eventgroup Ack entries referencing events and notification events that are transported via multicast shall reference an IPv4 Multicast Option and/or and IPv6 Multicast Option. The Multicast Options state to which Multicast address and port the events and notification events will be sent to. ⌋()

6.7.4.2.4 Subscribe Eventgroup Negative Acknowledgement (Subscribe Eventgroup Nack) Entry

[TR_SOMEIP_00393] ⌈ The Subscribe Eventgroup Negative Acknowledgment entry type shall be used to indicate that Subscribe Eventgroup entry was NOT accepted. ⌋*()*

[TR_SOMEIP_00394] ⌈ Subscribe Eventgroup Negative Acknowledgment entries shall set the entry fields in the following way:

- Type shall be set to 0x07 (SubscribeEventgroupAck).
- Service ID, Instance ID, Major Version, Eventgroup ID, Counter, and Reserved shall be the same value as in the Subscribe that is being answered.
- The TTL shall be set to 0x000000.

⌋*()*

[TR_SOMEIP_00566] ⌈ Reasons to not accept a Subscribe Eventgroup include (but are not limited to):

- Combination of Service ID, Instance ID, Eventgroup ID, and Major Version is unknown
- Required TCP-connection was not opened by client
- Problems with the references options occurred
- Resource problems at the Server

⌋*()*

[TR_SOMEIP_00527] ⌈ When the client receives a SubscribeEventgroupNack as answer on a SubscribeEventgroup for which a TCP connection is required, the client shall check the TCP connection and shall restart the TCP connection if needed. ⌋*()*

Rational:
The server might have lost the TCP connection and the client has not.

Checking the TCP connection might include the following:

- Checking whether data is received for e.g. other Eventgroups.
- Sending out a Magic Cookie message and waiting for the TCP ACK.
- Reestablishing the TCP connection.

6.7.5 Service Discovery Communication Behavior

6.7.5.1 Startup Behavior

[TR_SOMEIP_00395] ⌈ For each Service Instance or Eventgroup the Service Discovery shall have at least these three phases in regard to sending entries:

- Initial Wait Phase
- Repetition Phase
- Main Phase

⌋()

[TR_SOMEIP_00396] ⌈ An actual implemented state machine will need more than just states for these three phases. E.g. local services can be still down and remote services can be already known (no finds needed anymore). ⌋()

[TR_SOMEIP_00397] ⌈ As soon as the system has started and the link on a interface needed for a Service Instance is up (server) or requested (client), the service discovery enters the Initial Wait Phase for this service instance. ⌋()

[TR_SOMEIP_00398] ⌈ Systems has started means here the needed applications and possible external sensors and actuators as well. Basically the functionality needed by this service instance has to be ready to offer a service and finding a service makes only sense after some application already needs it. ⌋()

[TR_SOMEIP_00399] ⌈ The Service Discovery implementation shall wait based on the INITIAL_DELAY after entering the Initial Wait Phase and before sending the first messages for the Service Instance. ⌋()

[TR_SOMEIP_00400] ⌈ INITIAL_DELAY shall be defined as a minimum and a maximum delay. ⌋()

[TR_SOMEIP_00401] ⌈ The wait time shall be determined by choosing a random value between the minimum and maximum of INITIAL_DELAY. ⌋()

[TR_SOMEIP_00402] ⌈ The Service Discovery shall use the same random value for multiple entries of different types in order to pack them together for a reduced number of messages. ⌋()

[TR_SOMEIP_00403] ⌈ The Service Discovery shall also pack entries together, when no random delay is involved. For example shall all Subscribe Eventgroup entries of a message be answered together in one message. ⌋()

[TR_SOMEIP_00404] ⌈ After sending the first message the Repetition Phase of this Service Instance/these Service Instances is entered. ⌋()

[TR_SOMEIP_00405] ⌈ The Service Discovery implementation shall wait in the Repetitions Phase based on REPETITIONS_BASE_DELAY. ⌋()

[TR_SOMEIP_00406] ⌈ After each message sent in the Repetition Phase the delay is doubled. ⌋()

[TR_SOMEIP_00407] ⌈ The Service Discovery shall send out only up to REPETITIONS_MAX entries during the Repetition Phase. ⌋()

[TR_SOMEIP_00408] ⌈ Sending Find entries shall be stopped after receiving the corresponding Offer entries by jumping to the Main Phase in which no Find entries are sent. ⌋*()*

[TR_SOMEIP_00409] ⌈ If REPETITIONS_MAX is set to 0, the Repetition Phase shall be skipped and the Main Phase is entered for the Service Instance after the Initial Wait Phase. ⌋*()*

[TR_SOMEIP_00410] ⌈ After the Repetition Phase the Main Phase is being entered for a Service Instance. ⌋*()*

[TR_SOMEIP_00411] ⌈ After entering the Main Phase 1*CYCLIC_OFFER_DELAY is waited before sending the first message. ⌋*()*

[TR_SOMEIP_00412] ⌈ In the Main Phase Offer Messages shall be sent cyclically if a CYCLIC_OFFER_DELAY is configured. ⌋*()*

[TR_SOMEIP_00413] ⌈ After a message for a specific Service Instance the Service Discovery waits for 1*CYCLIC_OFFER_DELAY before sending the next message for this Service Instance. ⌋*()*

[TR_SOMEIP_00415] ⌈ For Find entries (Find Service and Find Eventgroup) no cyclic messages are allowed in Main Phase. ⌋*()*

[TR_SOMEIP_00582] ⌈ Requests/Subscriptions entries shall be triggered by Offer entries, which are sent cyclically. ⌋*()*

[TR_SOMEIP_00416] ⌈ Example:

Initial Wait Phase:

- Wait for random_delay in Range(INITIAL_DELAY_MIN, _MAX)
- Send message (Find Service and Offer Service entries)

Repetition Phase (REPETITIONS_BASE_DELAY=100ms, REPETITIONS_MAX=2):

- Wait $2^0 * 100ms$
- Send message (Find Service and Offer Service entries)
- Wait $2^1 * 100ms$
- Send message (Find Service and Offer Service entries)
- Wait $2^2 * 100ms$

Main Phase (as long message is active and CYCLIC_OFFER_DELAY is defined):

- Send message (Offer Service entries)
- Wait CYCLIC_OFFER_DELAY

⌋*()*

6.7.5.2 Server Answer Behavior

[TR_SOMEIP_00417] ⌈ The Service Discovery shall delay answers to entries that were transported in a multicast/broadcast SOME/IP-SD message using the configuration item REQUEST_RESPONSE_DELAY.

This applies to FindService entries.

⌋()

[TR_SOMEIP_00418] ⌈ The REQUEST_RESPONSE_DELAY shall also apply to unicast messages triggered by multicast messages (e.g. Subscribe Eventgroup after Offer Service). ⌋()

[TR_SOMEIP_00419] ⌈ The REQUEST_RESPONSE_DELAY shall not apply if unicast messages are answered with unicast messages. ⌋()

[TR_SOMEIP_00420] ⌈ REQUEST_RESPONSE_DELAY shall be specified by a minimum and a maximum. ⌋()

[TR_SOMEIP_00421] ⌈ The actual delay shall be randomly chosen between minimum and maximum of REQUEST_RESPONSE_DELAY. ⌋()

[TR_SOMEIP_00422] ⌈ For basic implementations all Find Service entries (no matter of the state of the Unicast Flag) shall be answered with Offer Service entries transported using unicast. ⌋()

[TR_SOMEIP_00423] ⌈ For optimization purpose the following behaviors shall be supported as option:

- Find messages received with the Unicast Flag set to 1 in main phase, shall be answered with a unicast response if the last offer was sent less than `1/2 CYCLIC_OFFER_DELAY` ago.
- Find messages received with the Unicast Flag set to 1 in main phase, shall be answered with a multicast RESPONSE if the last offer was sent `1/2 CYCLIC_OFFER_DELAY` or longer ago.
- Find messages received with Unicast Flag set to 0 (multicast), shall be answered with a multicast response. (Note: this was only needed in earlier migration scenarios and will go away in the future)

⌋()

6.7.5.3 Shutdown Behavior

[TR_SOMEIP_00427] ⌈ When a server service instance of an ECU is being stopped, a Stop Offer Service entry shall be sent out. ⌋()

[TR_SOMEIP_00428] ⌈ When a server sends out a Stop Offer Service entry all subscriptions for this service instance shall be deleted on the server side. ⌋()

[TR_SOMEIP_00429] ⌈ When a client receives a Stop Offer Service entry all subscriptions for this service instance shall be deleted on the client side. ⌋*()*

[TR_SOMEIP_00430] ⌈ When a client receives a Stop Offer Service entry, the client shall not send out Find Service entries but wait for Offer Service entry or change of status (application, network management, Ethernet link, or similar). ⌋*()*

[TR_SOMEIP_00431] ⌈ When a client service instance of an ECU is being stopped (i.e. the service instance is released), the SD shall send out Stop Subscribe Eventgroup entries for all subscribed Eventgroups. ⌋*()*

[TR_SOMEIP_00432] ⌈ When the whole ECUs is being shut down Stop Offer Service entries shall be sent out for all service entries and Stop Subscribe Eventgroup entries for Eventgroups. ⌋*()*

6.7.5.4 State Machines

[TR_SOMEIP_00433] ⌈ In this section the state machines of the client and server are shown. ⌋*()*

[TR_SOMEIP_00434] ⌈

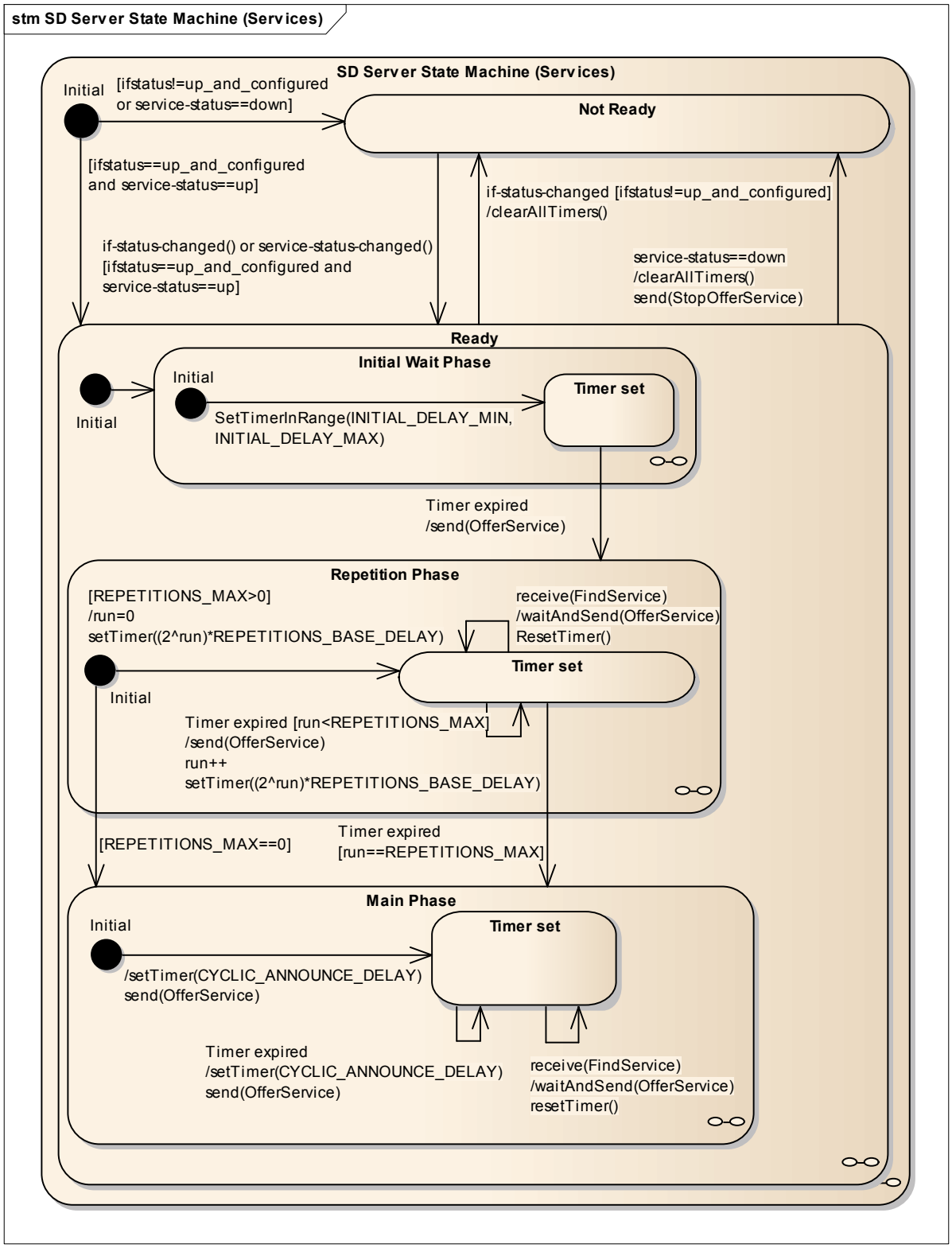

Figure 6.24: SOME/IP Service State Machine Server

⌋()

[TR_SOMEIP_00435] ⌈

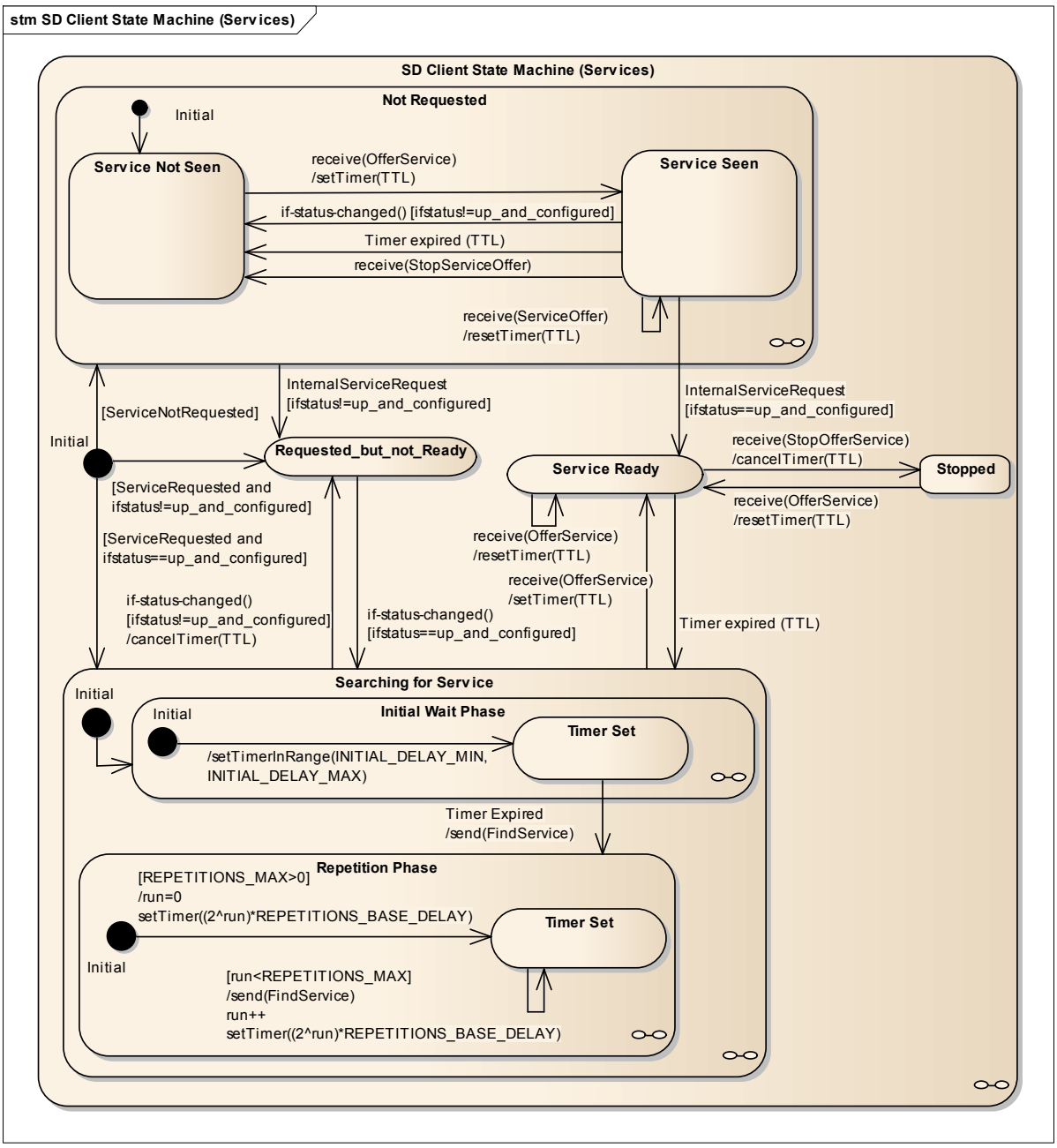

Figure 6.25: SOME/IP Service State Machine Client

]()

6.7.5.5 Error Handling

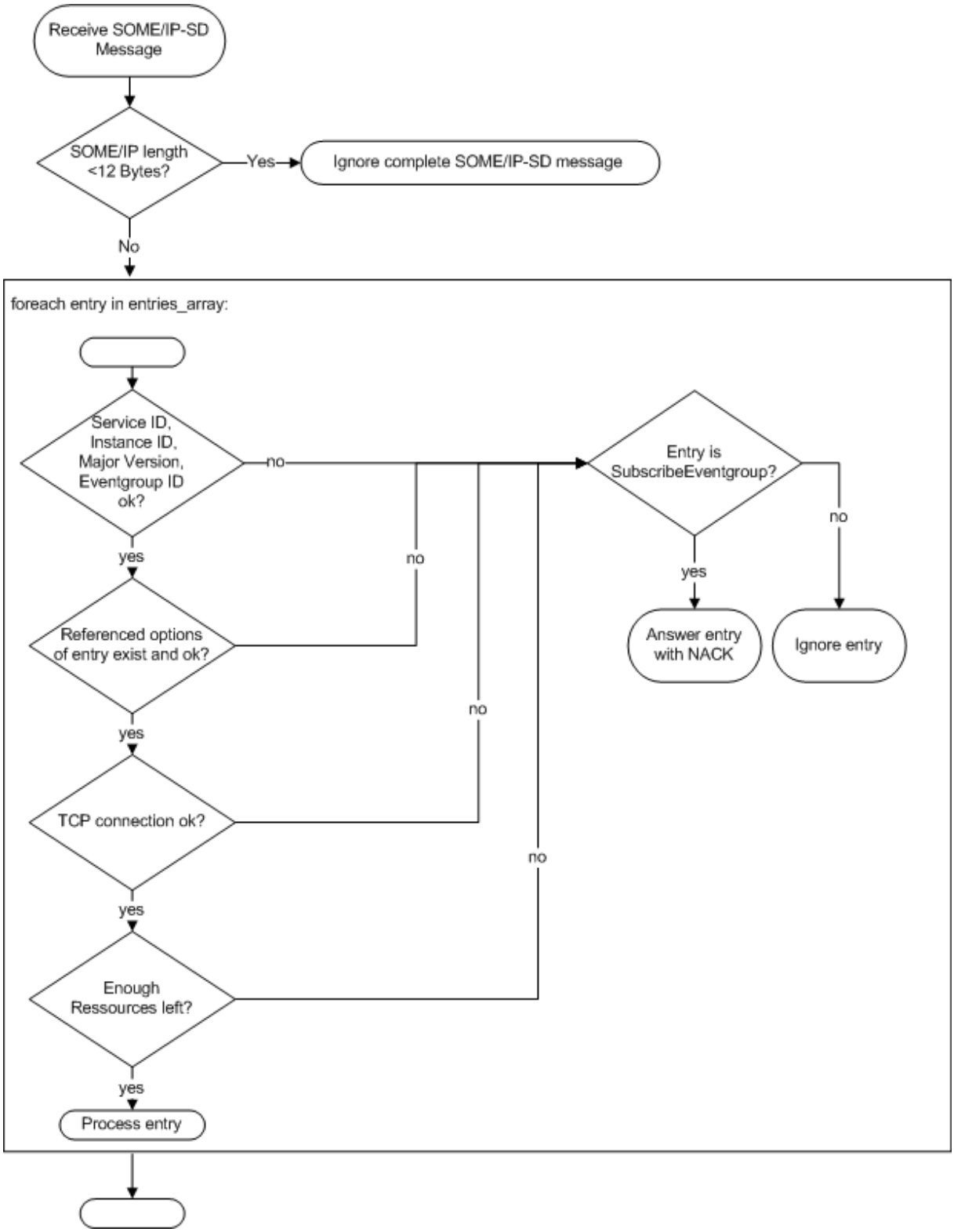

Figure 6.26: SOME/IP-SD Error Handling

Figure 6.26 shows a simplified process for the error handling of incoming SOME/IP-SD messages.

[TR_SOMEIP_00568] ⌈ The following steps shall be taken:

- Check that at least enough bytes for an empty SOME/IP-SD message are present.
- For each entry that can be parsed:
  - Check if the Service ID is known
  - Check if the Instance ID of this Service ID is known
  - Check if the Major Version of this Service Instance is known
  - Check if the Eventgroup ID of the Service Instance with Major Version is known (only applicable for eventgroup entries)
  - Check if the referenced Options exist in the options array and are syntactically ok
  - Check if the TCP connection is already present (only applicable, if TCP is configured for Eventgroup and Subscribe Eventgroup entry was received)
  - Check if enough ressources are left (e.g. Socket Connections)
- If at least one of these checks fails, you need to:
  - Answer with a Subscribe Eventgroup NACK, if the original entry was a Subscribe Eventgroup entry
  - Ignore, if the original entry was not a Subscribe Eventgroup entry

⌋()

6.7.6 Announcing non-SOME/IP protocols with SOME/IP-SD

[TR_SOMEIP_00436] ⌈ Besides SOME/IP other communication protocols are used within the vehicle; e.g. for Network Management, Diagnosis, or Flash Updates. Such communication protocols might need to communicate a service instance or have eventgroups as well. ⌋()

[TR_SOMEIP_00437] ⌈ For Non-SOME/IP protocols a special Service-ID shall be used and further information shall be added using the configuration option:

- Service-ID shall be set to 0xFFFE (reserved)
- Instance-ID shall be used as described for SOME/IP services and eventgroups.
- The Configuration Option shall be added and shall contain at least a entry with key "otherserv" and a configurable non-empty value that is determined by the system department.

⌋()

[TR_SOMEIP_00438] ⌈ SOME/IP services shall not use the otherserv-string in the Configuration Option. ⌋()

[TR_SOMEIP_00439] ⌈ For Find Service/Offer Service/Request Service entries the otherserv-String shall be used when announcing non-SOME/IP service instances. ⌋()

[TR_SOMEIP_00440] ⌈ Example for valid otherserv-string: "otherserv=internaldiag".
Example for a unvalid otherserv-string: "otherserv".
Example for a unvalid otherserv-string: "otherserv=". ⌋()

[TR_SOMEIP_00441] ⌈

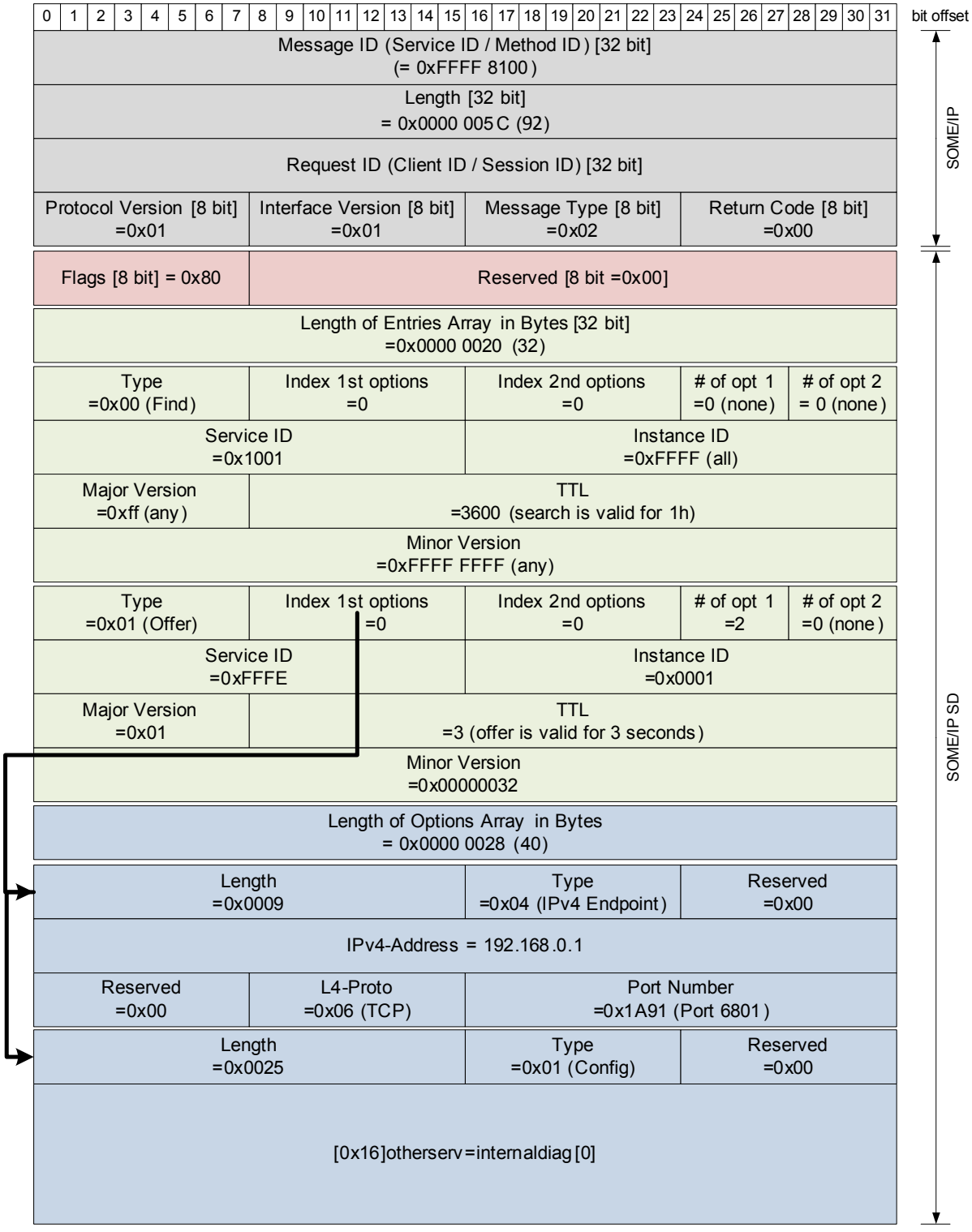

Figure 6.27: SOME/IP-SD Example PDU for Non-SOME/IP-SD

]()

6.7.7 Publish/Subscribe with SOME/IP and SOME/IP-SD

[TR_SOMEIP_00442] ⌈ In contrast to the SOME/IP request/response mechanism there may are cases in which a client requires a set of parameters from a server, but does not want to request that information each time it is required. These are called notifications and concern events and fields. ⌋*()*

[TR_SOMEIP_00443] ⌈ All clients needing events and/or notification events shall register using the SOME/IP-SD at run-time with a server. ⌋*()*

[TR_SOMEIP_00444] ⌈

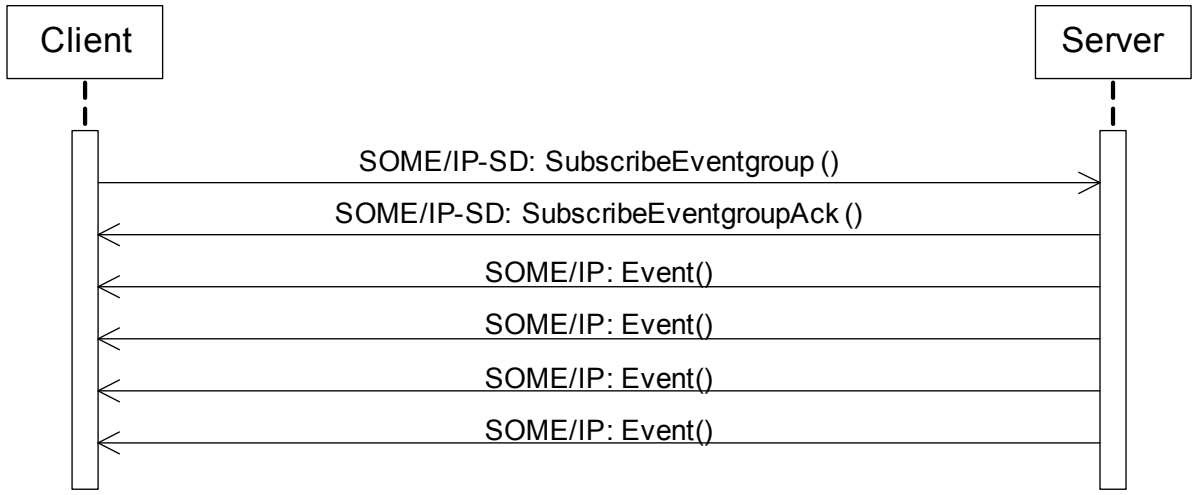

Figure 6.28: Notification interaction

⌋*()*

[TR_SOMEIP_00445] ⌈ This feature is comparable but NOT identical to the MOST notification mechanism. ⌋*()*

[TR_SOMEIP_00446] ⌈ With the SOME/IP-SD entry Offer Service the server offers to push notifications to clients; thus, it shall be used as trigger for Subscriptions. ⌋*()*

[TR_SOMEIP_00447] ⌈ When a server of a notification service starts up (e.g. after reset), it shall send a SOME/IP-SD Offer Service into the network to discover all instances interested in the events and fields offered. ⌋*()*

[TR_SOMEIP_00448] ⌈ Each client in SD based notification implements the specific service-interfaces for the notification they wish to receive and signal their wish of receiving such notifications using the SOME/IP-SD Subscribe Eventgroup entries. ⌋*()*

[TR_SOMEIP_00449] ⌈ Each client shall respond to a SOME/IP-SD Offer Service entry from the server with a SOME/IP-SD Subscribe Eventgroup entry as long as the client is still interested in receiving the notifications/events of this eventgroup.

If the client is able to reliably detect the reboot of the server using the SOME/IP-SD messages reboot flag, the client may choose to only answer Offer Service messages after the server reboots if configured to do so (TTL set to maximum value). The client

make sure that this works reliable even when the SOME/IP-SD messages of the server are lost. ⌋()

[TR_SOMEIP_00570] ⌈ If the client subscribes to two or more eventgroups including one or more identical events or fields, the server shall not send duplicated events or notification events for the field. This does mean regular events and not initial events. ⌋()

[TR_SOMEIP_00450] ⌈

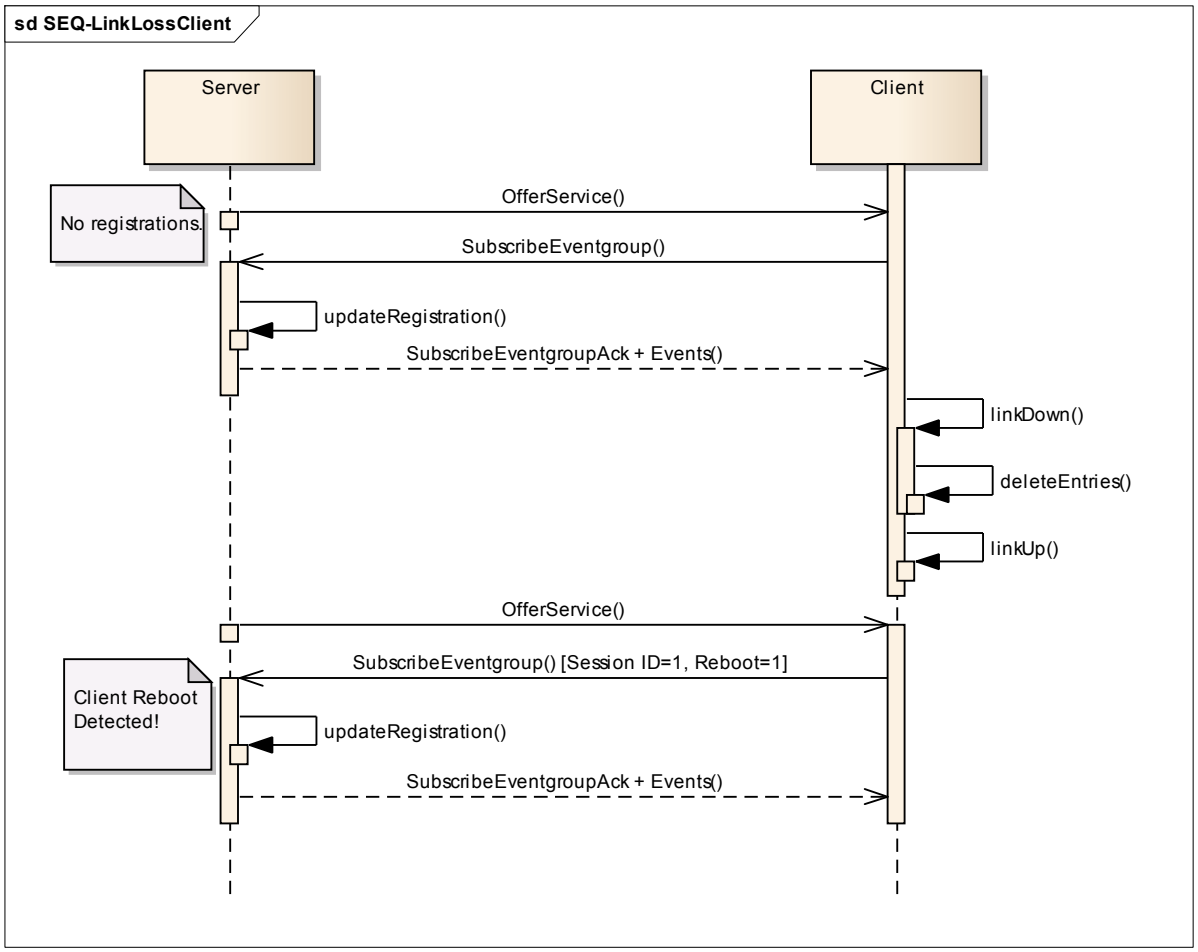

Figure 6.29: Publish/Subscribe with link loss at client (figure ignoring timings)

⌋()

[TR_SOMEIP_00451] ⌈ The server sending Offer Service entries as implicit Publishes has to keep state of Subscribe Eventgroup messages for this eventgroup instance in order to know if notifications/events have to be sent. ⌋()

[TR_SOMEIP_00452] ⌈ A client shall deregister from a server by sending a SOME/IP-SD Subscribe Eventgroup message with TTL=0 (Stop Subscribe Eventgroup). ⌋()

[TR_SOMEIP_00453] ⌈

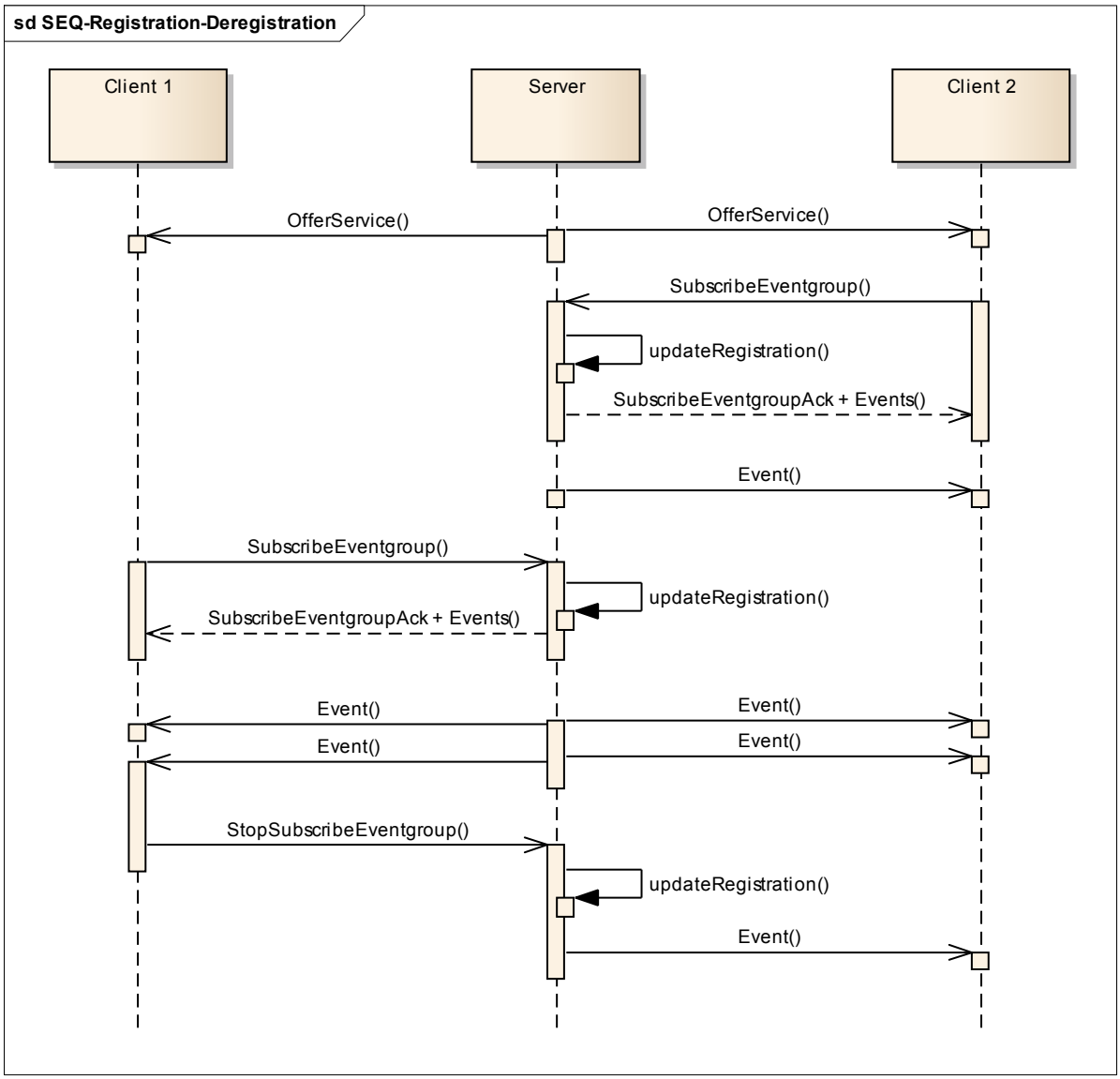

Figure 6.30: Publish/Subscribe Registration/Deregistration behavior (figure ignoring timings)

⌋()

[TR_SOMEIP_00454] ⌈ The SOME/IP-SD on the server shall delete the subscription, if a relevant SOME/IP error is received after sending the an event or notification event. ⌋()

The error includes but is not limited to not being able to reach the communication partner and errors of the TCP connection.

[TR_SOMEIP_00455] ⌈ If the server loses its link on the relavant Ethernet interface, it SHALL delete all the registered notifications and close the TCP connection for those notifications as well. ⌋()

[TR_SOMEIP_00456] ⌈ If the Ethernet link status of the server becomes up again, it shall trigger a SOME/IP-SD Offer Service message. ⌋()

[TR_SOMEIP_00457] ⌈

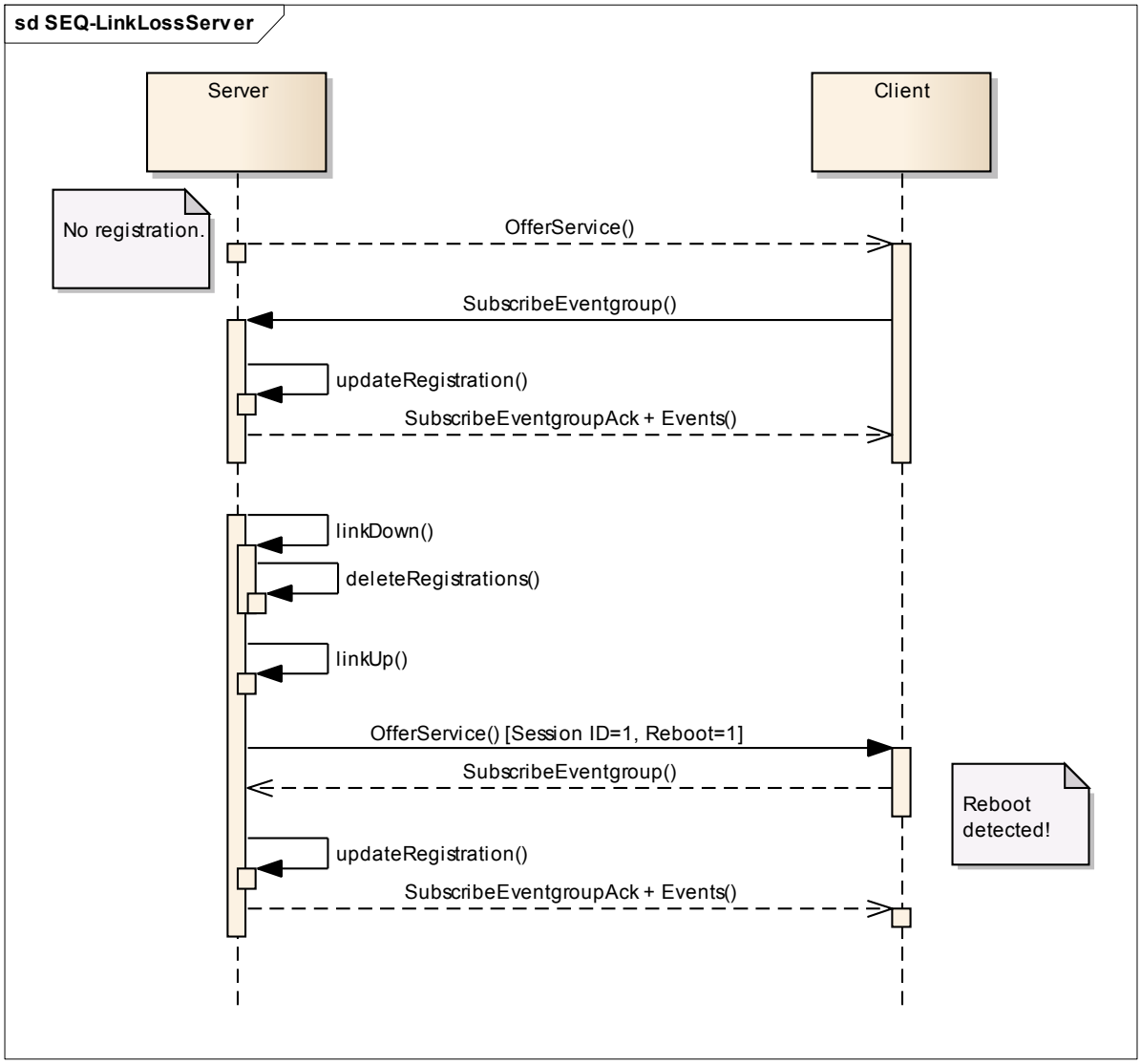

Figure 6.31: Publish/Subscribe with link loss at server (figure ignoring timings)

⌋()

[TR_SOMEIP_00458] ⌈ After having not received a notification/event of an eventgroup subscribed to for a certain timeout the ECU shall send a new Subscribe Eventgroup entry. The timeout shall be configurable for each eventgroup. ⌋()

[TR_SOMEIP_00459] ⌈ This timeout feature might be based on cyclic messages or message protected by Alive Counters (functional safety). ⌋()

[TR_SOMEIP_00460] ⌈ A link-up event on the clients Ethernet link shall start the Initial Wait Phase (consider UDP-NM and others). SOME/IP-SD Subscribe Eventgroup entry shall be sent out as described above. ⌋()

[TR_SOMEIP_00461] ⌈ The client shall open a TCP connection to the server before sending the Subscribe Eventgroup entry, if reliable events and notification events exist in the interface definition (e.g. FIBEX or ARXML). ⌋*()*

[TR_SOMEIP_00462] ⌈ After a client has sent a Subscribe Eventgroup entry the server shall send a Subscribe Eventgroup Ack entry considering the specified delay behavior. ⌋*()*

[TR_SOMEIP_00463] ⌈ The client shall wait for the Subscribe Eventgroup Ack entry acknowledging a Subscribe Eventgroup entry. If this Subscribe Eventgroup Ack entry does not arrive before the next Subscribe Eventgroup entry is sent, the client shall do the following: send a Stop Subscribe Eventgroup entry and a Subscribe Eventgroup entry in the SOME/IP-SD message the Subscribe Eventgroup entry would have been sent with. ⌋*()*

[TR_SOMEIP_00577] ⌈ The requirement [TR_SOMEIP_00463] shall not be applied to Offer Service entries that are a reaction to Find Service entries. This means that the Subscribe Eventgroup Ack entry of a Subscribe Eventgroup entry that was triggered by a unicast Offer Service entry is not monitored as well as upon a unicast Offer Service entry the Stop Subscribe Eventgroup entry/Subscribe Eventgroup entry is not sent. ⌋*()*

Note:
This behavior exists to cope with short durations of communication loss. The receiver of a Stop Subscribe Eventgroup and Subscribe Eventgroup combination would sent out Initial Events to lower the effects of the loss of messages.

[TR_SOMEIP_00464] ⌈ If the initial value is of concern - i.e. for fields - the server shall immediately send the first notification/event; i.e. event. The client shall repeat the Subscribe Eventgroup entry, if it did not receive the notification/event in a configurable timeout. ⌋*()*

[TR_SOMEIP_00465] ⌈ This means:

- It is not allowed to send initial values of events upon subscriptions (pure event and not field).
- The event messages of field notifiers shall be sent on subscriptions (field and not pure event).
- If a subscription was already valid and is updated by a Subscribe Eventgroup entry, no initial events shall be sent.
- Receiving Stop Subscribe / Subscribe combinations trigger initial events of field notifiers.

⌋*()*

[TR_SOMEIP_00466] ⌈

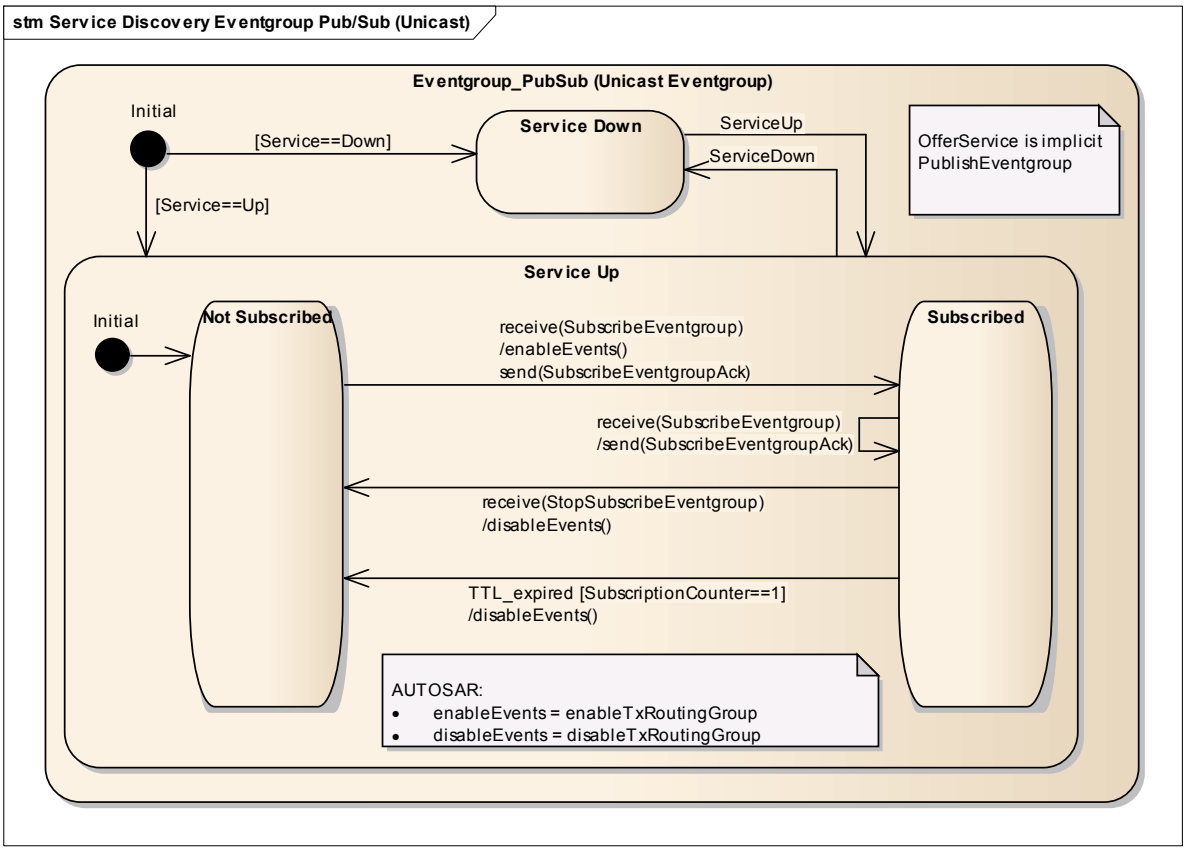

Figure 6.32: Publish/Subscribe State Diagram (server behavior for unicast eventgroups)

⌋()

[TR_SOMEIP_00571] ⌈ If a client subscribes to different eventgroups of the same Service Instance that all include the same field in different SOME/IP-SD messages, the Server shall send out the initial events for this field for every subscription separately. ⌋()

[TR_SOMEIP_00572] ⌈ If a client subscribes to different eventgroups of the same Service Instance that all include the same field in the same SOME/IP-SD message, the Server may choose to not send out the initial event for this field more than once. ⌋()

Note:
This means the Server can optimize by sending the initial events only once, if supported by its architecture.

[TR_SOMEIP_00467] ⌈

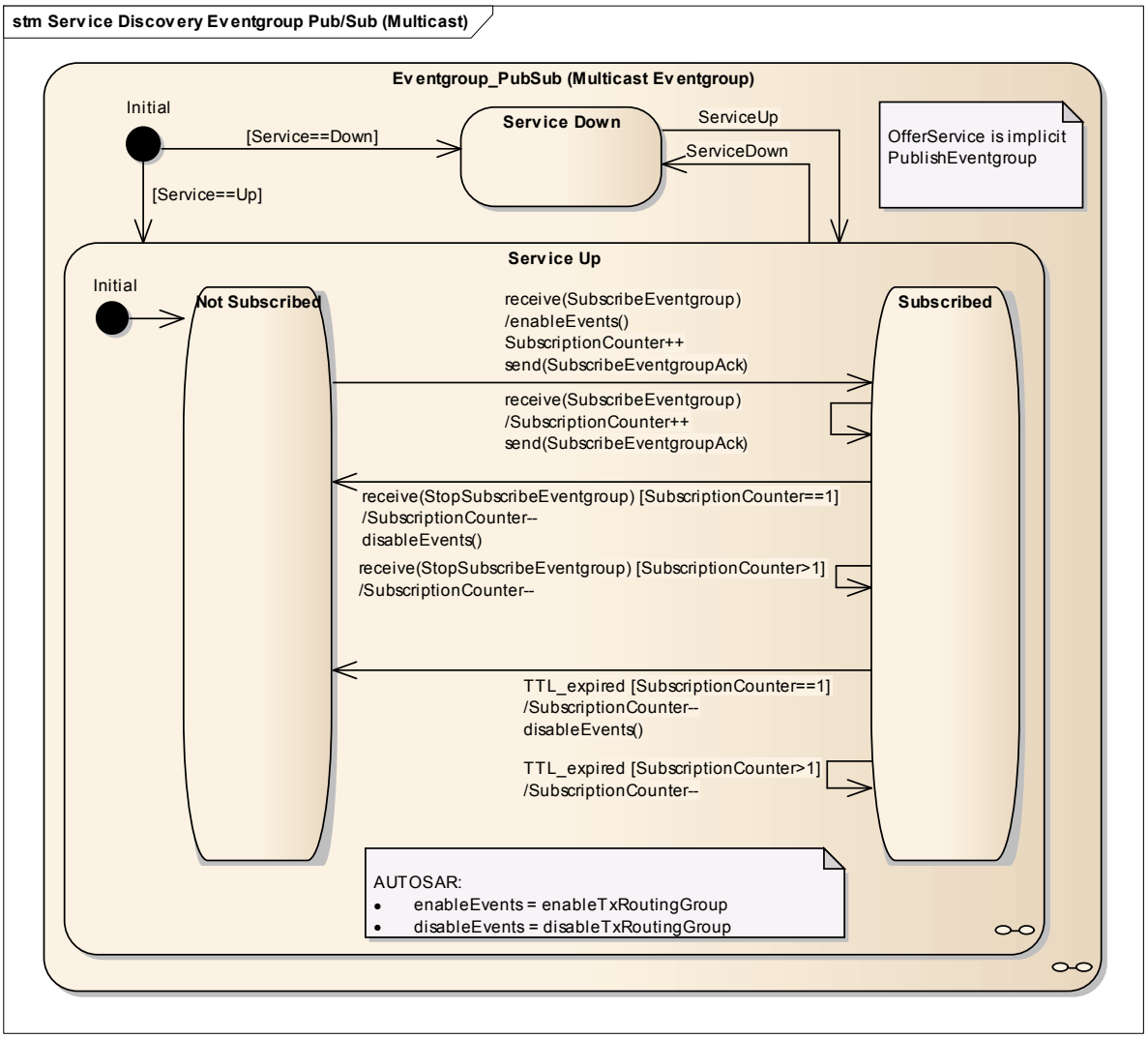

Figure 6.33: Publish/Subscribe State Diagram (server behavior for multicast eventgroups)

⌋()

[TR_SOMEIP_00468] ⌈

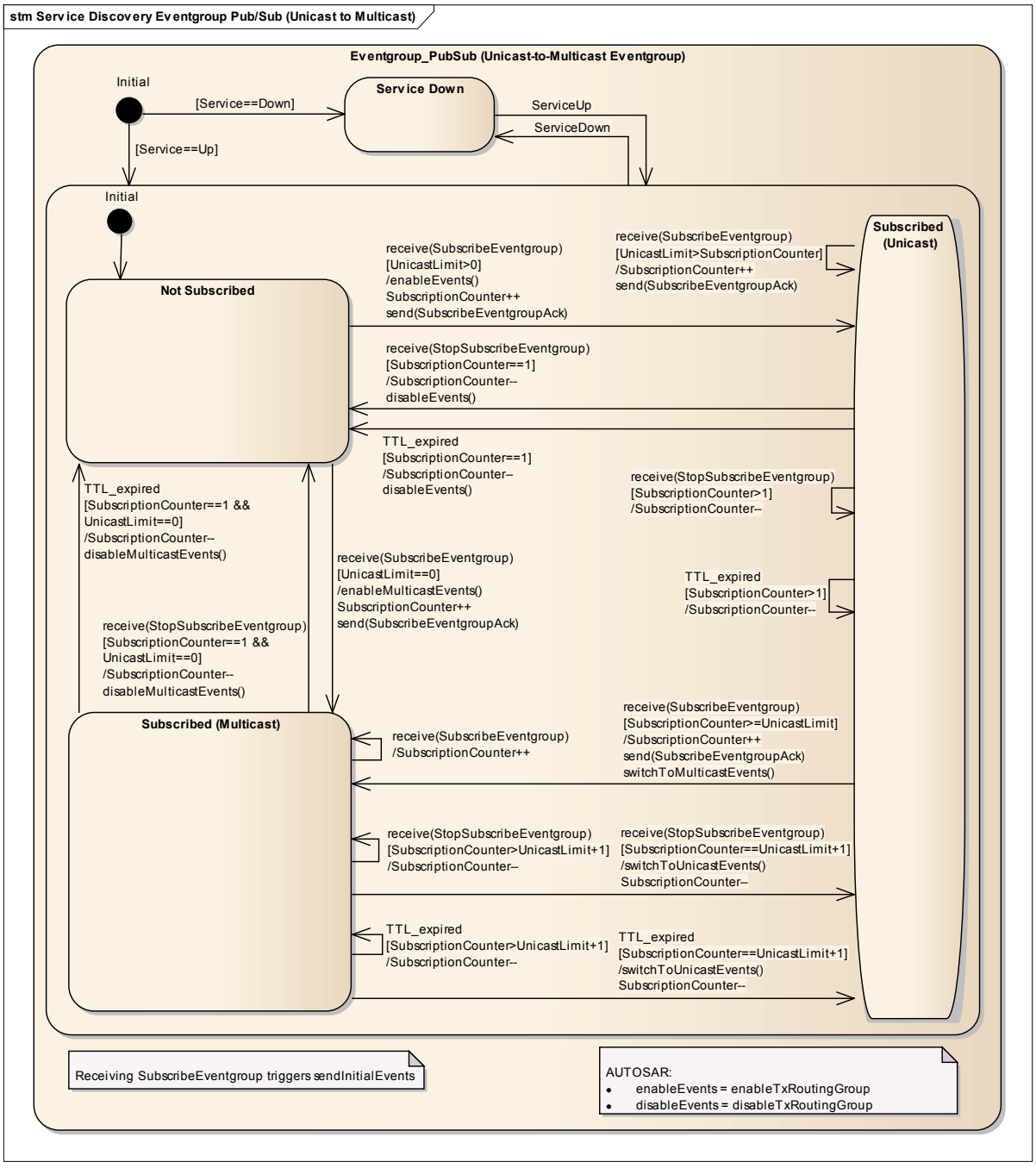

Figure 6.34: Publish/Subscribe State Diagram (server behavior for adaptive unicast/multicast eventgroups)

⌋()

[TR_SOMEIP_00469] ⌈

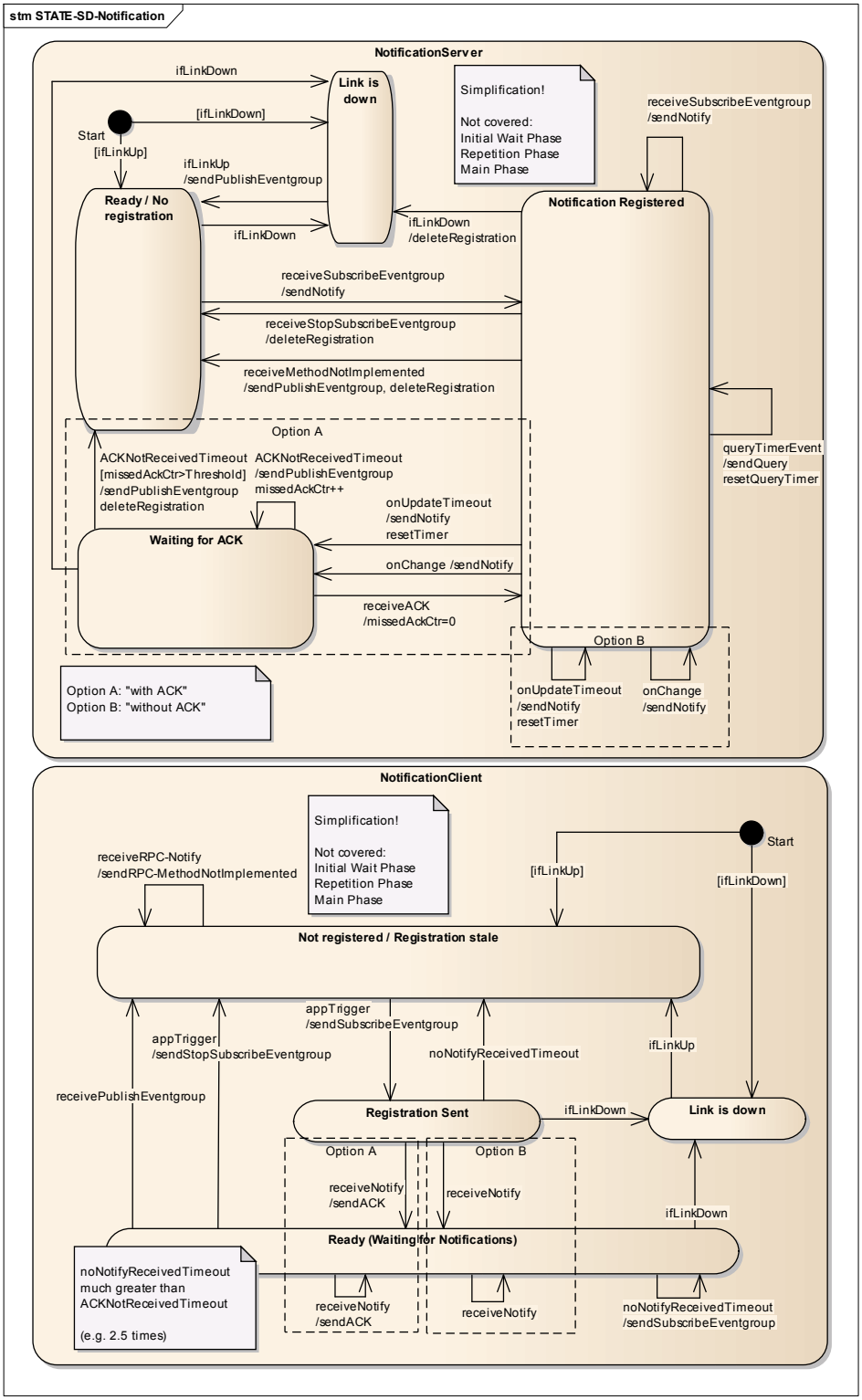

Figure 6.35: Publish/Subscribe State Diagram (overall behavior)

⌋()

[TR_SOMEIP_00470] ⌈ An implicit registration of a client to receive notifications from a server shall be supported. Meaning the mechanism is pre-configured. ⌋()

[TR_SOMEIP_00471] ⌈ To allow for cleanup of stale client registrations (to avoid that the list of listeners fills over time), a cleanup mechanism is required. ⌋*()*

[TR_SOMEIP_00472] ⌈ The following entries shall be transported by unicast only:

- Subscribe Eventgroup
- Stop Subscribe Eventgroup
- Subscribe Eventgroup Ack
- Subscribe Eventgroup Nack

⌋*()*

[TR_SOMEIP_00473] ⌈ When sending a Subscribe Eventgroup entry as reaction of receiving an Offer Service entry, the timer controlling cyclic Subscribe Eventgroups entries shall be reset. ⌋*()*

[TR_SOMEIP_00474] ⌈ If no cyclic Subscribe Eventgroups are configured, the timer for cyclic Subscribe Eventgroups stays turned off. ⌋*()*

6.7.8 Endpoint Handling for Services and Events

[TR_SOMEIP_00475] ⌈ This section describes how the Endpoints encoded in the Endpoint and Multicast Options shall be set and used. ⌋*()*

[TR_SOMEIP_00476] ⌈ The Service Discovery shall overwrite IP Addresses and Port Numbers with those transported in Endpoint and Multicast Options if the statically configured values are different from those in these options. ⌋*()*

6.7.8.1 Service Endpoints

[TR_SOMEIP_00477] ⌈ Offer Service entries shall reference up to 1 UDP Endpoint Option and up to 1 TCP Endpoint Option. Both shall be of the same version Internet Protocol (IPv4 or IPv6). ⌋*()*

[TR_SOMEIP_00478] ⌈ The referenced Endpoint Options of the Offer Service entries denote the IP Address and Port Numbers the service instance is reachable at the server. ⌋*()*

[TR_SOMEIP_00479] ⌈ The referenced Endpoint Options of the Offer Service entries also denote the IP Address and Port Numbers the service instance sends the events from. ⌋*()*

[TR_SOMEIP_00480] ⌈ Events of this service instance shall not be sent from any other Endpoints than those given in the Endpoint Options of the Offer Service entries. ⌋*()*

[TR_SOMEIP_00481] ⌈ If an ECU offers multiple service instances, SOME/IP messages of these service instances shall be differentiated by the information transported in the Endpoint Options referenced by the Offer Service entries. ⌋*()*

[TR_SOMEIP_00482] ⌈ Therefore transporting an Instance ID in the SOME/IP header is not required. ⌋*()*

[TR_SOMEIP_00528] ⌈ A sender shall not reference Endpoint Options nor Multicast Options in a Find Service Entry. ⌋*()*

[TR_SOMEIP_00529] ⌈ A receiver shall ignore Endpoint Options and Multicast Options in a Find Service Entry. ⌋*()*

[TR_SOMEIP_00530] ⌈ Other Options (neither Endpoint nor Multicast Options), shall still be allowed to be used in a Find Service Entry. ⌋*()*

6.7.8.2 Eventgroup Endpoints

[TR_SOMEIP_00483] ⌈ Subscribe Eventgroup entries shall reference up to 1 UDP Endpoint Option and up to 1 TCP Endpoint Option for the Internet Protocol used (IPv4 or IPv6). ⌋*()*

[TR_SOMEIP_00484] ⌈ The Endpoint Options referenced in the Subscribe Eventgroup entries is also used to send unicast UDP or TCP SOME/IP events for this Service Instance. ⌋*()*

[TR_SOMEIP_00485] ⌈ Thus the Endpoint Options referenced in the Subscribe Eventgroup entries are the IP Address and the Port Numbers on the client side. ⌋*()*

[TR_SOMEIP_00486] ⌈ TCP events are transported using the TCP connection the client has opened to the server before sending the Subscribe Eventgroup entry. See Chapter 6.7.3.4.3. ⌋*()*

[TR_SOMEIP_00487] ⌈ The initial events shall be transported using unicast from Server to Client. ⌋*()*

[TR_SOMEIP_00488] ⌈ Subscribe Eventgroup Ack entries shall reference up to 1 Multicast Option for the Internet Protocol used (IPv4 or IPv6). ⌋*()*

[TR_SOMEIP_00489] ⌈ The Multicast Option shall be set to UDP as transport protocol. ⌋*()*

[TR_SOMEIP_00490] ⌈ The client shall open the Endpoint specified in the Multicast Option referenced by the Subscribe Eventgroup Ack entry as fast as possible to not miss multicast events. ⌋*()*

[TR_SOMEIP_00491] ⌈ If the server has to send multicast events very shortly (configurable time < 5 ms) after sending the Subscribe Eventgroup Ack entry, the server shall try to delay these events, so that the client is not missing it. If this event was sent as multicast anyhow, the server shall send this event using unicast as well. ⌋*()*

6.7.8.3 Example

[TR_SOMEIP_00492] ⌈ Figure 6.36 shows an example with the different Endpoint and a Multicast Option:

- The Server offers the Service Instance on Server UDP-Endpoint SU and Server TCP-Endpoint ST
- The Client opens a TCP connection
- The Client sends a Subscribe Eventgroup entry with Client UDP-Endpoint CU (unicast) and a Client TCP-Endpoint CT.
- The Server answers with a Subscribe Eventgroup Ack entry with Multicast MU

Then the following operations happen:

- The Client calls a method on the Server
- Request is sent from CU to SU and Response from SU to CU
- For TCP this would be: Request dyn to ST and RESPONSE from ST to CT
- The Server sends a Unicast UDP Event: SU to CU
- The Server sends a Unicast TCP Event: ST to CT
- The Server sends a Multicast UDP Event: SU to MU

Keep in mind that Multicast Endpoints use a Multicast IP Address on the receiver side, i.e. the client, and TCP cannot be used for Multicast communication.

⌋()

[TR_SOMEIP_00493] ⌈

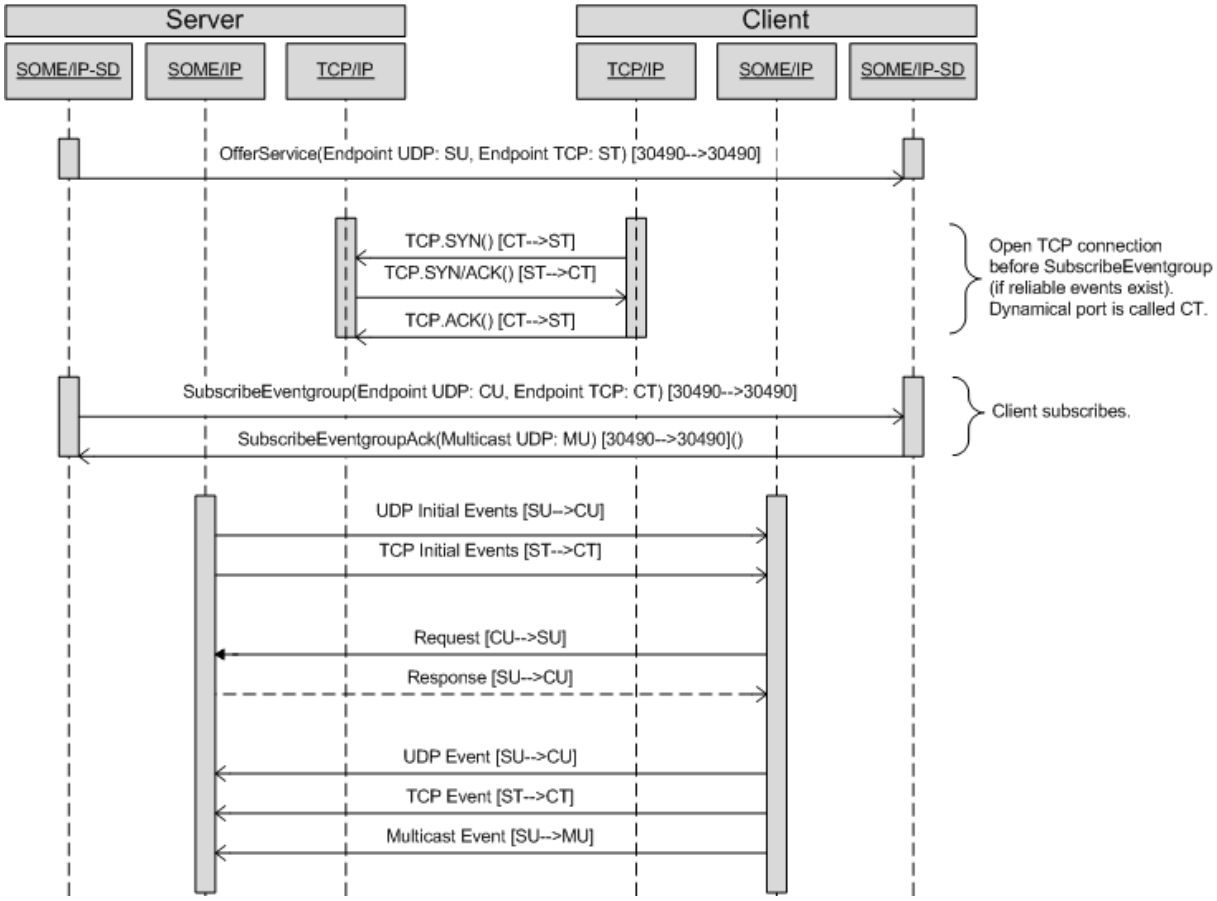

Figure 6.36: Publish/Subscribe Example for Endpoint Options and the usage of ports

⌋()

6.7.9 Mandatory Feature Set and Basic Behavior

[TR_SOMEIP_00494] ⌈ In this section the mandatory feature set of the Service Discovery and the relevant configuration constraints are discussed. This allow for bare minimum implementations without optional or informational features that might not be required for current use cases. ⌋()

[TR_SOMEIP_00495] ⌈ The following information is defined as compliance check list(s). If a feature is not implemented, the implementation is considered not to comply to SOME/IP-SDs basic feature set. ⌋()

[TR_SOMEIP_00496] ⌈ The following entry types shall be implemented:

- Find Service
- Offer Service
- Stop Offer Service
- Subscribe Eventgroup

- Stop Subscribe Eventgroup
- Subscribe Eventgroup Ack
- Subscribe Eventgroup Nack

⌋()

[TR_SOMEIP_00497] ⌈ The following option types shall be implemented, when IPv4 is required:

- IPv4 Endpoint Option
- IPv4 Multicast Option
- Configuration Option
- IPv4 SD Endpoint Option (receiving at least)

⌋()

[TR_SOMEIP_00498] ⌈ The following option types shall be implemented, if IPv6 is required:

- IPv6 Endpoint Option
- IPv6 Multicast Option
- Configuration Option
- IPv6 SD Endpoint Option (receiving at least)

⌋()

[TR_SOMEIP_00499] ⌈ The following option types shall be implemented, if non-SOME/IP services or additional configuration parameters are required:

- Configuration Option

⌋()

[TR_SOMEIP_00500] ⌈ The following behaviors/reactions shall be implemented on the Server side:

- The Server shall offer services including the Initial Wait Phase, the Repetition Phase, and the Main Phase depending on the configuration.
- The Server shall offer services using Multicast (Repetition Phase and Main Phase).
- The Server does not need to answer a Find Service in the Repetition Phase.
- The Server shall answer a Find Service in the Main Phase with an Offer Service using Unicast (no optimization based on unicast flag).
- The Server shall send a Stop Offer Service when shutting down.

- The Server shall receive a Subscribe Eventgroup as well as a Stop Subscribe Eventgroup and react according to this specification.
- The Server shall send a Subscribe Eventgroup Ack and Subscribe Eventgroup Nack using unicast.
- The Server shall support controlling the sending (i.e. fan out) of SOME/IP event messages based on the subscriptions of SOME/IP-SD. This might include sending events based on Multicast.
- The Server shall support the triggering of initial SOME/IP event messages.

⌋()

[TR_SOMEIP_00501] ⌈ The following behaviors/reactions shall be implemented on the Client side:

- The Client shall find services using a Find Service entry and Multicast only in the repetition phase.
- The Client shall stop finding a service if the regular Offer Service arrives.
- The Client shall react to the Servers Offer Service with an unicast SD message that includes all Subscribe Eventgroups of the services offered in the message of the Server that the client currently wants to subscribe to.
- The Client shall interpret and react to the Subscribe Eventgroup Ack and Subscribe Eventgroup Nack as specified in this document.

⌋()

[TR_SOMEIP_00502] ⌈ The following behavior and configuration constraints shall be supported by the Client:

- The Client shall even to handle Eventgroups if only the TTL of the SD Timings is specified. This means that of all the timings for the Initial Wait Phase, the Repetition Phase, and the Main Phase only TTL is configured. This means the client shall only react on the Offer Service by the Server.
- The Client shall answer to an Offer Service with a Subscribe Eventgroup even without configuration of the Request-Response-Delay, meaning it should not wait but answer instantaneously.

⌋()

[TR_SOMEIP_00503] ⌈ The Client and Server shall implement the Reboot Detection as specified in this document and react accordingly. This includes but is not limited to:

- Setting Session ID and Reboot Flag according to this specification.
- Keeping a Session ID counter only used for sending Multicast SD messages.
- Keeping Session ID counters for every Unicast relation for sending Unicast SD messages.

- Understanding Session ID and Reboot Flag according to this specification.
- Keeping a Multicast Session ID counter per ECU that exchanges Multicast SD messages with this ECU.
- Keeping a Unicast Session ID counter per ECU that exchanges Unicast SD messages with this ECU.
- Detecting reboot based on this specification and reaction accordingly.
- Correctly interpreting the IPv4 and IPv6 SD Endpoint Options in regard to Reboot Detection.

⌋()

[TR_SOMEIP_00504] ⌈ The Client and Server shall implement the "Endpoint Handling for Service and Events". This includes but is not limited to:

- Adding 1 Endpoint Option UDP to an Offer Services if UDP is needed.
- Adding 1 Endpoint Option TCP to an Offer Service if TCP is needed.
- Adding 1 Endpoint Option UDP to Subscribe Eventgroup if events over UDP are required.
- Adding 1 Endpoint Option TCP to Subscribe Eventgroup if events over TCP are required.
- Adding 1 Multicast Option UDP to Subscribe Eventgroup Ack if multicast events are required.
- Understanding and acting according to the Endpoint and Multicast Options transported as described above.
- Overwriting preconfigured values (e.g. IP Addresses and Ports) with the information of these Endpoint and Multicast Options.
- Interpreting incoming IPv4 and IPv6 Endpoint Options as SD endpoints instead of the Address and Port number in the outer layers.

⌋()

[TR_SOMEIP_00573] ⌈ The Client and Server which implement the "SD Endpoint option" shall interpret the incoming IPv4 and IPv6 Endpoint Options as SD endpoints instead of the Address and Port number in the outer layers. ⌋()

[TR_SOMEIP_00583] ⌈ The following SOME/IP-SD Option types in relation to Entry types are allowed:

	Endpoint	**Multicast**	**Configuration**	**Load Balancing**
FindService	0	0	0-1	0-1
OfferService	1-2	0	0-1	0-1

StopOffer Service	1-2	0	0-1	0-1
Subscribe Eventgroup	1-2	0	0-1	0-1
StopSubscribe Eventgroup	1-2	0	0-1	0-1
Subscribe EventgroupAck	0	0-1	0-1	0-1
Subscribe Eventgroup	0	0	0-1	0-1

⌋()

6.7.10 SOME/IP-SD Mechanisms and Errors

[TR_SOMEIP_00505] ⌈ In this section SOME/IP-SD in cases of errors (e.g. lost or corrupted packets) is discussed. This is also be understood as rationale for the mechanisms used and the configuration possible. ⌋()

[TR_SOMEIP_00506] ⌈ Soft State Protocol: SOME/IP-SD was designed as soft state protocol, that means that entries come with a lifetime and need to be refreshed to stay valid (setting the TTL to the maximum value shall turn this off). Using cyclic Offer Service entries and the TTL as aging mechanism SOME/IP-SD shall cope with many different cases of errors. Some examples:

- If a client or server leaves without sending a Stop entry or this Stop entry got lost, the system will fix itself after the TTL expiration.
- If an Offer Service entry does not arrive because the packet got lost, the system will tolerate this based on the value of the TTL.

Example configuration parameter for fast healing: cyclic delays 1s and TTL 3s. ⌋()

[TR_SOMEIP_00507] ⌈ Initial Wait Phase:

The Initial Wait Phase was introduced for two reasons: deskewing events of starting ECUs to avoid traffic bursts and allowing ECUs to collect multiple entries in SD messages. ⌋()

[TR_SOMEIP_00508] ⌈ Repetition Phase:

The Repetition Phase was introduced to allow for fast synchronization of clients and servers. If the clients startup later, it will find the server very fast. And if the server starts up later, the client can be found very fast. The Repetition Phase increases the time between two messages exponentially to avoid that overload situtaions keep the system from synchronization.

An example configuration could be REPETITIONS_BASE_DELAY=30ms and REPETITIONS_MAX=3. ⌋()

[TR_SOMEIP_00509] ⌈ Main Phase:

In the Main Phase the SD tries to stabilize the state and thus decreases the rate of packets by sending no Find Services anymore and only offers in the cyclic interval (e.g. every 1s). ⌋()

[TR_SOMEIP_00510] ⌈ Request-Response-Delay:

SOME/IP-SD shall be configured to delay the answer to entries in multicast messages by the Request-Response-Delay (in FIBEX called Query-Response-Delay). This is useful in large systems with many ECUs. When sending a SD message with many entries in it, a lot of answers from different ECUs arrive at the same time and put large stress on the ECU receiving all these answers. ⌋()

6.8 Migration and Compatibility

6.8.1 Supporting multiple versions of the same service.

[TR_SOMEIP_00511] ⌈ In order to support migrations scenarios ECUs shall support serving as well as using different incompatible versions of the same service. ⌋()

[TR_SOMEIP_00512] ⌈ In order to support a Service with more than one version the following is required:

- The server shall offer the service instance of this service once per major version.
- The client shall find the service instances once per supported major version or shall use the Major Version as 0xFF (all versions).
- The client shall subscribe to events of the service version it needs.
- All SOME/IP-SD entries shall use the same Service-IDs and Instance-IDs but different Major Versions.
- The server has to demultiplex messages based on the socket it arrives, the Message-ID, and the Major Versions.

⌋()

[TR_SOMEIP_00513] ⌈ For AUTOSAR supporting more than one version might mean that serialization and deserialization might have to be done by an serializer or proxy component. ⌋()

6.9 Reserved and special identifiers for SOME/IP and SOME/IP-SD.

In this chapter an overview of reserved and special identifiers are shown.

[TR_SOMEIP_00515] ⌈ Reserved and special Service-IDs:

Service-ID	Description
0x0000	Reserved
0xFF00 - 0xFF1F	Reserved for Testing at OEM
0xFF20 - 0xFF3F	Reserved for Testing at Tier-1
0xFF40 - 0xFF5F	0xFF5F Reserved for ECU Internal Communication (Tier-1 proprietary)
0xFFFE	Reserved for announcing non-SOME/IP service instances.
0xFFFF	SOME/IP and SOME/IP-SD special service (Magic Cookie, SOME/IP-SD, ...).

⌋()

[TR_SOMEIP_00516] ⌈ Reserved and special Instance-IDs:

Instance-ID	Description
0x0000	Reserved
0xFFFF	All Instances

⌋()

[TR_SOMEIP_00517] ⌈ Reserved and special Method-IDs/Event-IDs:

Method-ID	Description
0x0000	Reserved
0x7FFF	Reserved
0x8000	Reserved
0xFFFF	Reserved

⌋()

[TR_SOMEIP_00531] ⌈ Reserved eventgroup-IDs:

Eventgroup-ID	Description
0x0000	Reserved
0xFFFF	All Eventgroups

⌋()

[TR_SOMEIP_00519] ⌈ Method-IDs and Event-IDs of Service 0xFFFF:

Method-ID/Event-ID	Description
0x0000	SOME/IP Magic Cookie Messages
0x8000	SOME/IP Magic Cookie Messages
0x8100	SOME/IP-SD messages (events)

⌋()

[TR_SOMEIP_00520] ⌈ Besides "otherserv" other names are supported by the configuration option. The following list gives an overview of the reserved names:

Name	Description
hostname	Used to name a host or ECU.
instancename	Used to name an instance of a service.
servicename	Used to name a service.
otherserv	Used for non-SOME/IP Services.

⌋()