Matthias Stefan, Bachelor of Engeneering

Fragmentation of the Realtime Acyclic protocol in PROFINET

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Software Development and Business Management

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Otto Koudelka

Institute of Communication Networks and Satellite Communications
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Otto Koudelka

Graz, March 2018

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Acknowledgment

I want to to say thank you to my supervisor Martin Meissnitzer who answered all the questions I had and read through the thesis several times for correction. Thank you Thomas Pössler for reading through my thesis and helping my with the integration of the protocol into the PROFINET norm. Thank you Ines Mohlzan, Herbert Tanner and Gregor Mang who made it possible that I could write this thesis at Siemens AG. Thank you to all my other colleagues in Graz that supported me all the time.

Thank you Berta and Resi (names of coffee machines) for your hot liquid that helped me to continue working. Unfortunately Berta is no longer with us. Thanks to my friend Thilo Graßmann, who had a sympathetic ear for me when I made no progress with my thesis. Thank you to all my friends who gave me the right balance after work and thanks to my parents for the support during the whole master program.

Last but not least thank you Marika Scheer for managing the official stuff with the university.

Abstract

PROFINET is an industrial Ethernet protocol with support for real time transportation. The real time acyclic (RTA) protocol is used in version 1 to send alarms. In version 2 it should support transportation of PROFINET payloads for start up and parameterization, which is called Remote Service Interface (RSI). The payloads (records) exceed the maximum payload length of RTA. This is why version 2 of RTA needs fragmentation and reassembly functionality.

Version 2 is designed based on the comparison and analysis of fragmentation strategies of existing protocols (TCP, IPv4, IPv6, SCTP, IKEv2 and RPC). Before RPC was used to transport records. Three variants are described. All support in order transportation of a maximum of 16 MBytes of payload with frame drop detection. The basic variant uses a separated communication channel for the alarms to transport records. The advanced variant uses one channel and shares its receive resources for records and alarms. The chunk variant transports both in one Ethernet packet. Packet definition, state machines and sequence diagrams are designed for all variants.

The basic variant has been implemented. PROFINET communication relations can be established. Retransmission mechanisms and resource use have optimized in contrast to the RPC implementation.

Compared to RPC the calls are transported 15% faster with RSI. The additional memory usage of the test application with enabled RSI is 4 KBytes. For now RPC and IP components are not removable and the code size increased by about 10 Kbytes. Once they are, the code size can be reduced by about 321 KBytes.

Contents

Abstract vii

Index of abbreviations xv

1. Introduction 1

2. Motivation and Objectives 3

2.1. System Architecture 3

2.2. Data transmission 5

2.3. Software Architecture 7

3. Fragmentation strategies 9

3.1. Naming 9

3.2. TCP 10

3.3. IPv4 12

3.3.1. Fragmentation Procedure 14

3.3.2. Reassembling Procedure 15

3.4. IPv6 16

3.4.1. Fragment Header 18

3.4.2. Fragmentation Process 18

3.4.3. Reassembling 19

3.5. IKEv2 19

3.5.1. Fragmentation Process 23

3.5.2. Reassembling 23

3.6. SCTP 24

3.6.1. Fragmentation and Reassembly Procedure 27

3.6.2. (Selective) Acknowledgment Mechanism 27

3.7. RPC 29

3.7.1. Fragmentation Process 32

3.8. Comparison 34
3.8.1. Packet Structure 36
3.8.2. Sequences 41
3.8.3. Flow Control 48
3.8.4. Memory 51

4. Protocol Design **55**
4.1. Packet Structure 55
4.1.1. PDU Type 56
4.1.2. Flags 58
4.1.3. Fragmentation Information 59
4.1.4. Implementations 61
4.2. Sequence 64
4.2.1. Basic Variant 65
4.2.2. Advanced Variant 67
4.2.3. Advanced Variant with Chunks 74
4.3. State Machine 78
4.3.1. Basic Variant 78
4.3.2. Advanced Variant 83
4.3.3. Chunk Variant 85

5. Implementation **87**
5.1. Test Interface 89
5.2. Frame Observation 89
5.3. Time 93
5.4. Memory and Code Size 95

6. Conclusion **97**

Bibliography **99**

Appendix **103**

A. RTA Fragmentation Variant Basic - Sequence **105**

B. RTA Fragmentation Variant Advanced - Sequence **113**

C. Initiator State Machine of the Basic Variant. 117

D. Responder State Machine of the Basic Variant. 123

E. State Machine of the Advanced Variant 131

F. LMPM State Machine 135

G. Comparison Fragmentation 141

H. Comparison Memory 143

List of Figures

2.1. PROFINET Logical Typology 4
2.2. NRT and RT Channel of Profinet IO 6
2.3. Software Stack 7

3.1. TCP Header in Byte Definition 10
3.2. Flags in TCP 11
3.3. Definition of IPv4 Header. 13
3.4. Definition of IPv6 Header. 17
3.5. Definition of IKEv2 Encrypt Frag Header 21
3.6. Structure of IKEv2 packet 22
3.7. Structure of SCTP Packet 25
3.8. Structure of SCTP Selective Acknowledgment Chunk 28
3.9. Header of Connectionless RPC 30
3.10. Fragment-ACK Payload 33
3.11. RTA Header Version 1 37
3.12. Comparison Fragmentation Bits 38
3.13. PROFINET RPC Sequence Diagram 42
3.14. Sequence Diagram SCTP 45
3.15. Sequence Diagram IKEv2 45
3.16. Sequence Diagram TCP 46
3.17. RPC Fragment Sequence Diagram 46
3.18. TCP Without Intersection 49
3.19. TCP With Intersection 49
3.20. SCTP Call With Intersection 50
3.21. Comparison of Memory Used for Fragmentation 52

4.1. Changes of RTA Header Version 2 56
4.2. Flags Implementation Possibilities 59
4.3. Fragmentation Header Realization Possibilities. 60

4.4. Packet Variant 1: RTA with Fix RSI Extension 62
4.5. Packet Variant 2: Optional Header Realization. 63
4.6. Packet Variant 3: Chunk Variant 63
4.7. Example Sequence Diagram 1 RSI Fragments Basic Variant . 66
4.8. Framedrops Request 68
4.9. Example Sequence Diagram of RSI Fragments Basic Variant . 69
4.10. Example Sequence Diagram of Advanced Variant 71
4.11. Example Sequence Diagram of Advanced Variant With Alarms and Records 72
4.12. Server Sends Alarm During RSI Send Process 73
4.13. Example Sequence Diagram of Advanced Variant With Overlap 2 75
4.14. Example Sequence Diagram of Advanced Variant With Chunks 76
4.15. Initiator State Machine of Basic Variant 79
4.16. Responder State Machine of Basic Variant 82
4.17. Alarm State Machine for Advanced Variant 84

5.1. Development Environment 88
5.2. PCIOX GUI Without Connection 90
5.3. PCIOX GUI After Connection Phase 91
5.4. Wireshark Connect 92
5.5. Wireshark Fragmented Request 93
5.6. Compare Fragmentation 94
5.7. Code Size in Bits 96

Index of abbreviations

ACK	Acknowledgment
ACP	Alarm Consumer Provider
AP	Access Point
CLRPC	Connectionless Remote Procedure Call
CM	Context Management
EDD	Ethernet Device Driver
EPM	Endpoint Mapper
IKE	Internet Key Exchange
Ind	Indication
IO	In Out
ISN	Initial Sequence Number
IV	Initialization Vector
MTU	Maximum Transmission Unit
NRT	Non Real Time
Req	Request
RPC	Remote Procedure Call
RQB	Request Block
RSI	Remote Service Interface
Rsp	Response

Index of abbreviations

RT	Real Time
RTA	Real Time Acyclic
PDU	Protocol Description Unit
SACK	Selective Acknowledgment
SAP	Service Access Point
SCTP	Stream Control Transmission Protocol
TCP	Transport Control Protocol
TTL	Time To Live
TSN	Time Sensitive Network
UDP	User Datagram Protocol

1. Introduction

PROFINET is an industrial protocol based on Ethernet. It is defined in IEC61158 and IEC61784 as an open standard. (PROFIBUS & PROFINET International, 2016, p. 1) At the end of 2016 there have been 16,4 million devices used and supported. (PROFIBUS & PROFINET International, 2017[b]) There are three fields of application: factory automation, process automation and motion control. They are used in manufacturing facilities for cars, food & beverages, planes, medicine and many more (PROFIBUS & PROFINET International, 2017[a]) to connect, monitor and control the production process.

The PROFINET software is separated into different components that are placed in different layers. There are different classes of transportation. PROFINET communication is using Ethernet Frames. For its Content Management data which is not time critical IP and UDP stacks are included. Parallel to that there is a communication channel for realtime applications that sit on top of the Data Link Layer. The Realtime Acyclic (RTA) protocol is used for transporting alarms in realtime.

In the status quo the UDP/IP stack is included in every device that runs PROFINET. In order to increase the stability of PROFINET the new protocol does not necessarily use the UDP/IP stacks. The resource management of IP must not influence the stability. In order to be more flexible and to build devices, that are cheaper and more stable it is intended to build devices, which will not contain an IP/UDP stack. Profinet@TSN is an IEEE standard that gives more robustness to non realtime IP communication. As a next step PROFINET is intended to be integrated into Time-Sensitive Network (TSN) in Layer 2. A further step would be to remove IP completely. (Henning, 2017)

PROFINET uses Remote Procedure Calls (RPC) for communication initialization and parameterization. RPC uses IP as network layer and UDP for transportation. The new version will replace these two layers with the RTA protocol. In order to port the RPCs to the new version of RTA it needs to be able to transport the maximum payload of RPC which is 64 KBytes. The maximum payload of RTA is 1432 bytes. This will be extended through a fragmentation procedure.

A research of existing fragmentation strategies and analysis of the RTA protocol will be made. As output it will provide several solutions for implementing fragmentation of RTA and the embedding of RPC calls in the PROFINET component Alarm Consumer Provider (ACP). ACP is the component where RTA is implemented and the alarms are processed. The most promising solution of this research will be designed in detail. In order to find the best solution a deep understanding of the system architecture and structure is necessary, which will be described in the following chapter.

2. Motivation and Objectives

PROFINET is an open industrial standard and protocol that is based on Ethernet. It is developed by the PROFIBUS user organization. The protocol supports realtime and non realtime communication. (Popp, 2005, p. 12). PROFINET has been divided into two parts: PROFINET CBA for modular plant manufacturing and PROFINET IO for connecting decentral periphery. PROFINET CBA is not used anymore. PROFINET IO is used for this thesis.

This section describes the covered PROFINET stack, the system and software architecture and the planned porting of Remote Procedure Calls to the ACP component. It shows a prospect of the benefits that are expected concerning to code size and performance, starting with the most outer one.

2.1. System Architecture

The PROFINET system architecture uses Controllers and Devices that are connected via Ethernet. Controller and Device are set up in a Client / Server model. The controller is the client that initiates the connection to the the device which is the server. The Controller provides output data and consumes input data and the device provides input data and consumes output data. The Controller initializes a connection to a device and parameterizes it. Next it analyzes diagnosis information from the device. (Bormann and Hilgenkamp, 2006, pp. 199,200)

The PROFINET typology and the inner structure that represents the addressing model of an IO device or an IO controller are displayed in Figure 2.1. All communication participants are connected via an Ethernet bus. The

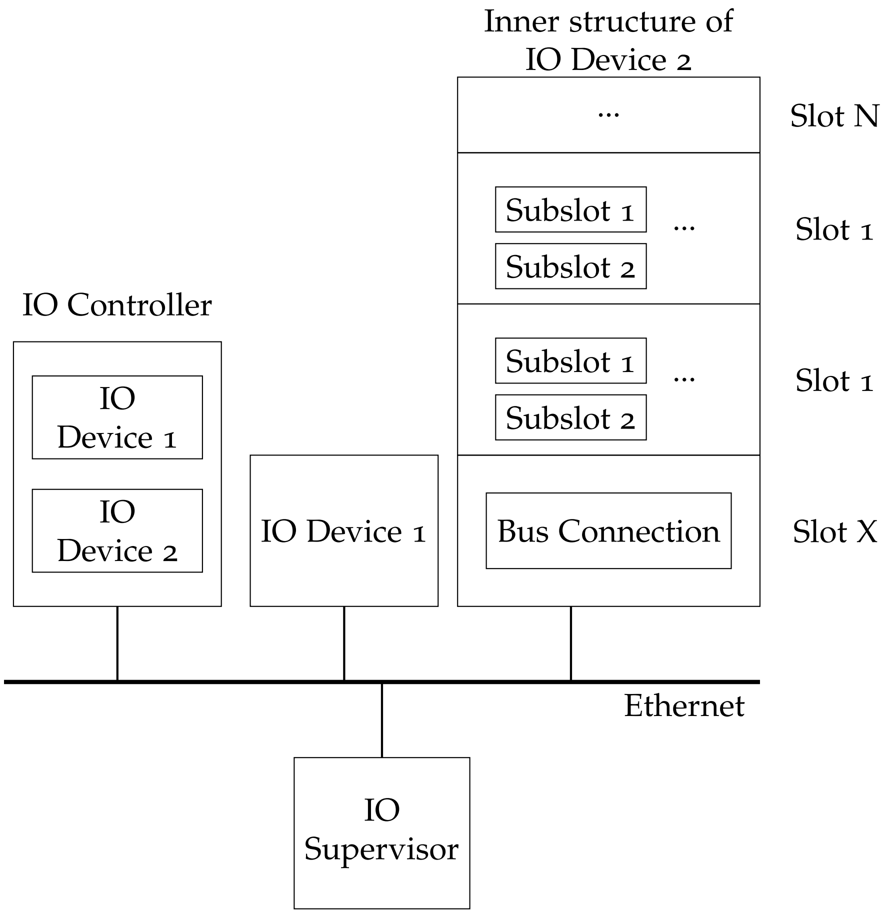


Figure 2.1.: Profinet Logical Typology. IO Controller and Devices are connected via Ethernet. One controller may control one or more devices. One device is controllable by one or more controllers. The controller has the structure for every connected device inside. The IO supervisor is an engineering station and starts the commissioning and searches for failures. (Popp, 2005, pp. 41,42,43)

devices and controllers are modeled with slots and subslots, which represent physical modules that can be pulled and plugged. Each module may contain one subslot which is the interface to the processes. The slot of the bus connections is decided through the projection. (Popp, 2005, pp. 42,43)

One controller is connected to one or more devices. It is also possible that one device is connected to more than one controller which is necessary for redundancy, fail safe and shared-device functionalities. A shared device is controlled by two or more controllers. The IO Supervisor is an engineering station for failure diagnosis and configuring the whole plant. The Totally Integrated Automaton (TIA) Portal software (Siemens AG, 2017) controls a production plant and is used to design the plant virtually and produces configuration files for the startup. This configuration includes the definition and description of the subslots. (Popp, 2005, pp. 42,43)

In the physical connection between a controller and devices there are different classes of communication within PROFINET. They differ basically in their latency.

2.2. Data transmission

PROFINET uses two different ways of data communication that are displayed in Figure 2.2. One is used for realtime applications and the other for non realtime applications. The non realtime applications use an IP stack for routing and UDP as the transport protocol. This is the standard Ethernet communication and it is used for establishing a connection between a controller and a device, reading diagnosis information, parameterizing the device and configuring the IP address over PROFINET DCP (Henning, 2015). These are defined in the PROFINET services Connect, Release, Read, Write, Control and Read Implicit. The standard Ethernet communication does provide realtime functionality. PROFINET adds this functionality in parallel. The Ethertype $0x8892$ is reserved for PROFINET. Within PROFINET the Frame-ID is a description of the payload. The value $0xFE01$ identifies the RTA protocol version 1 and the Frame-ID of $0xFE02$ will identify the RTA protocol verison 2 which can additionally transport Remote Service

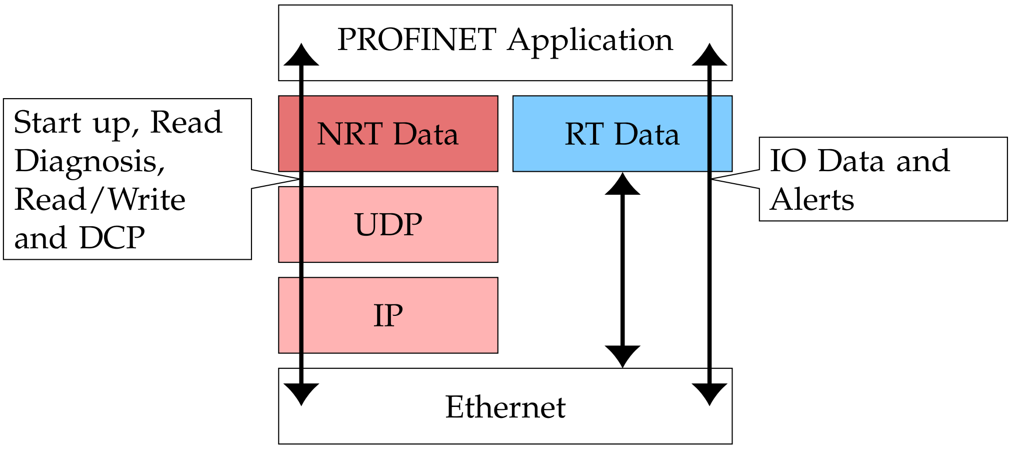


Figure 2.2.: NRT and RT channel of Profinet IO. Based on the Ethernet frame the communication is divided into NRT and RT channels. NRT is directed through the IP- and UDP stacks. RT Data is directly delivered to the PROFINET application. RT data has the EtherType $0x8892$. NRT data has the Ethertypes $0x8000$ or $0x0806$. (Popp, 2005, p. 48)

Interface **RSI** data as payload. RSI is the name for the new functionality that transports Remote Procedure Calls over RTA.

The realtime communication is used for IO data and alarms. The realtime communication can be extended to an *isochronous realtime* (IRT) communication, which guarantees cycle synchronization of cyclic IO data. IRT does not have to be part of a PROFINET device. (Popp, 2005, pp. 47,48)

The goal of this thesis is to route the PROFINET services over RTA instead of the NRT data path. The data does not have to be processed by the IP and UDP stacks. These stacks do not have to be compiled in devices or controllers and the required memory space is reduced. The software is separated in components. The structure and layout of the architecture is described in the following section.

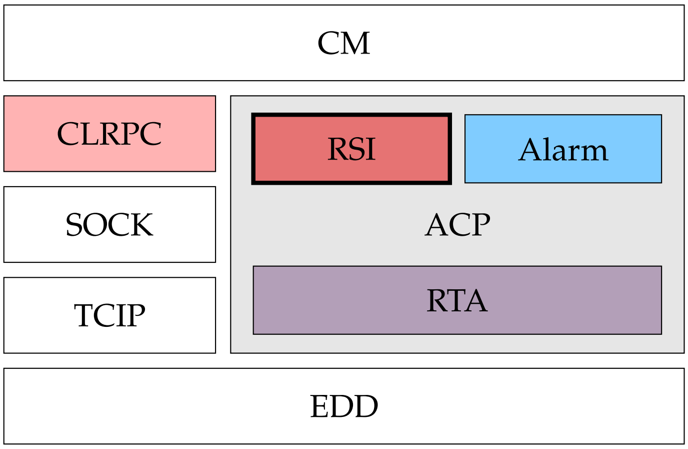


Figure 2.3.: Software stack with record services without IP. Displayed are the PROFINET components and ACPs inner component. The NRT ressourses are directed to TCIP and the realtime packets (RTA) are directed to the ACP component. RTA shall now transport the realtime alarms and also the record services in RSI. RSI takes over the services to CLRPC. CM controls ACP and CLRPC (if it is included). The Ethernet Device Driver (EDD) is below all components.

2.3. Software Architecture

Figure 2.3 displays the part of the PROFINET stack which is used to transport records over the PROFINET component Connectionless RPC **CLRPC** and the alarms that are processed by Alarm Consumer Provicder **ACP** over the realtime channel RT. Furthermore it shows the upcoming feature Remote Service Interface **RSI** that will transport the records with RTA for transportation. The service elements that are currently transported by CLRPC will be transported with RSI.

The current payload of **RTA** is limited to 1432 bytes. A data record may have a maximum size of 64 KBytes. ACP has to fragment and reassemble the records. The choice of the right fragmentation strategy is based on the transport protocol **RTA** and the system architecture where it is embedded.

UDP over TCIP

The component **TCIP**, displayed in Figure 2.3 includes the TCP/IP and also **UDP/IP**. TCP is not used to send records.

To support the transportation of records in RSI the existing RTA protocol needs to be fragmented. To find a suitable fragmentation strategy a research have been made, as described in the following chapter.

3. Fragmentation strategies

Other protocol standards are using fragmentation of their packets. This chapter shall analyze the most important, well known and established protocols and their fragmentation strategies. The protocols will be analyzed regarding their other capabilities and for which use cases they are designed. The extra features like flow control, in order transmission, header size, flexibility, routing, OSI-Layer and used memory will be pointed out.

The basic fragmentation strategy and the extra features are the basis for this comparison and the recommendation for a realization. Therefore advantages and disadvantages are pointed out. A common naming convention is necessary for that as described below.

3.1. Naming

Each protocol may use different words for the same meaning. So the following terms are used instead of the ones that are defined. This makes the comparison easier to understand.

- **Header**: Contains additional information (like routing, fragmentation).
- **Fragmentation Header**: Part of the header that contains the fragmentation information, if it is separated in an own header.
- **Data**: The data from an upper level that has to be transported.
- **Payload**: Whole or partial data that is sent in a packet or fragment.
- **Packet**: Contains header and payload.
- **Fragment**: Describes a packet if the data is fragmented.
- **Chunk**: Describes a packet (may be fragmented) that is the data of a higher protocol part, optionally together with other queued chunks.

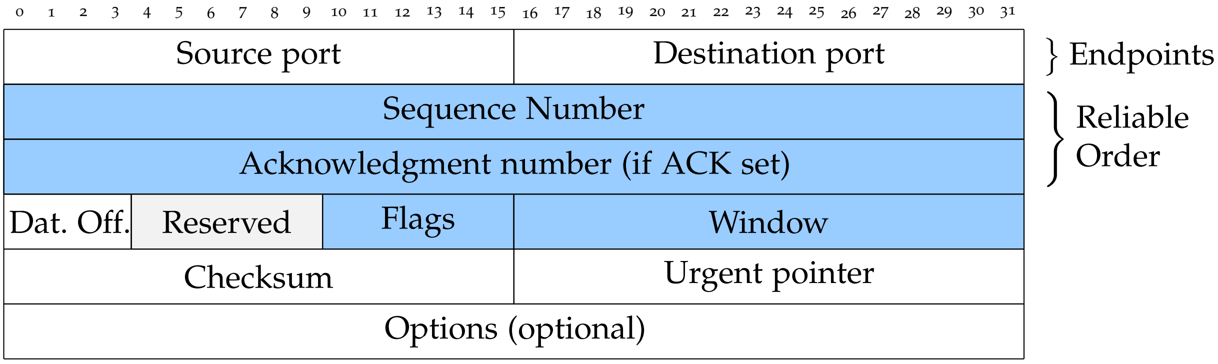


Figure 3.1.: Definition of the TCP Header. It includes endpoints for routing; sequence number and acknowledgment number, for the transmission in the right order and for checking what has been received; Flags as control mechanism; WindowSize for flow control and a checksum for error detection.
🟦 : Used for fragmentation process.

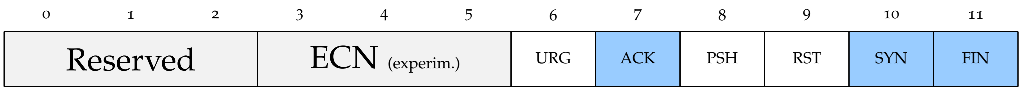
- **Call**: A call is a message orientated transmission of data from one node to another.
- **Request**: A request is a call by an initiator.
- **Response**: A response is a transmission of data in reaction to an incoming request.

These terms are used to describe the structure and behavior of the following protocols that support fragmentation. A detailed analysis follows.

3.2. TCP

The Transmission Control Protocol is a commonly used byte stream protocol that sends data from one node to another. TCP includes error-checking, ordered transmission and reliability of the data. It does not differentiate between an initiator or a responder and there is no request or response on this layer. There is a bidirectional transportation of bytes.

The TCP header is defined in Figure 3.1. ***Source port*** and ***Destination port*** are the endpoints of the communication within the hosts. The calling host has to know the destination port of the called host for the transmission.



0	1	2	3	4	5	6	7	8	9	10	11
Reserved			ECN (experim.)			URG	ACK	PSH	RST	SYN	FIN

Figure 3.2.: Flags in TCP with the initial verion from 1981 and the current version:

- ECN: Flags for Explicit Congestion Notification for signal an upcumming overload of the router (in ip nets). Experimental feature.
- URG: Urgentpointer used
- ACK: Acknowledgment Number used
- PSH: Ask for push buffer to application
- RST: Reset connection
- SYN: Synchronize Sequence Numbers
- FIN: Last packet from sender

🟦 : Used for fragmentation process.

Standard ports for TCP are well known and depends on the upper protocol, e.g. HTTP uses 80, HTTPS 443, XMPP 5222 and 5269. (IANA, 2017[b]) The port maps the service. If there is no standard port defined for a certain higher protocol/service a unused number can be assigned by used applications.

Sequence Number addresses the packet(s) that have been sent by the caller. The ***Acknowledgment Number*** is set by the receiver to tell the caller which sequence number it is expecting next. The numbers are connected to each other and the sequence number starts with the initial value called ISN. The establishment of a connection and how the sequence and acknowledgment numbers are used is described in Section 3.8.2. (RFC 793, 1980)

The ***Data Offset*** describes the length of the TCP header in 32 bit words. This includes the Options field which is an optional header with extra connection information. The Data Offset indicates where the payload starts.

The ***Flags*** part of the protocol is defined in Figure 3.2. The Flags are used for connection establishment **SYN**, acknowledgment of received data **ACK**, resetting the connection **RST**, ending **FIN**, telling the receiving host how many bytes are left and demanding for immediate transport of the received data to the application **PSH**.

TCP supports flow control with the ***Window*** part. The receiving host tell the sender how many bytes it can be received until the next acknowledgment.

The ***Checksum*** includes an error check over the header and the payload. If it is used over IP it also includes some fields like IP addresses from the IP header.

The ***Urgent Pointer*** is an offset from the current sequence number to the last sequence number.

Fragmentation as a Base

The fragmentation mechanism is a core feature of TCP. The whole protocol supports a reliable transportation of a bytestream with an arbitrary length which is built upon network packets. In contrast to the other strategies this one is not message orientated. Messages are built the layer above. This message may coincide with a TCP connection but it do not have to. In HTTP 0.9, which is used not longer, every HTTP request started a new TCP connection and closed it after finishing. In HTTP 1.0 the keep alive function made it possible to transport multiple HTTP requests in one TCP connection. In HTTP 1.1 it became the default variant. (RFC 2616, 1999)

The state of the art use of TCP for internet communication it uses one TCP connection for many requests.

3.3. IPv4

IP is used for routing packets in networks through IP-Addresses.

Internet Protocol Version 4 **IPv4** supports fragmentation. The fragmentation is used for transmissions in ‘small packet‘ networks. (RFC 760, 1981, p. 2). IP packets can be sent through many different network parts (nets). One net is specified from one router to another. Every net may have different Maximum Transmission Units **MTU**s. The packet may be fragmented.

The IP header is displayed in Figure 3.3. The Version is $0x4$. The ***IHL***, Internet Header Length, describes the header length in 32 bit words. The defined minimal value is 5. The technical maximum value is 60 bytes. If the header length is not dividable by 32 the header length is padded. The ***Type of Service*** specifies and categorize the data that is transported: precedence,

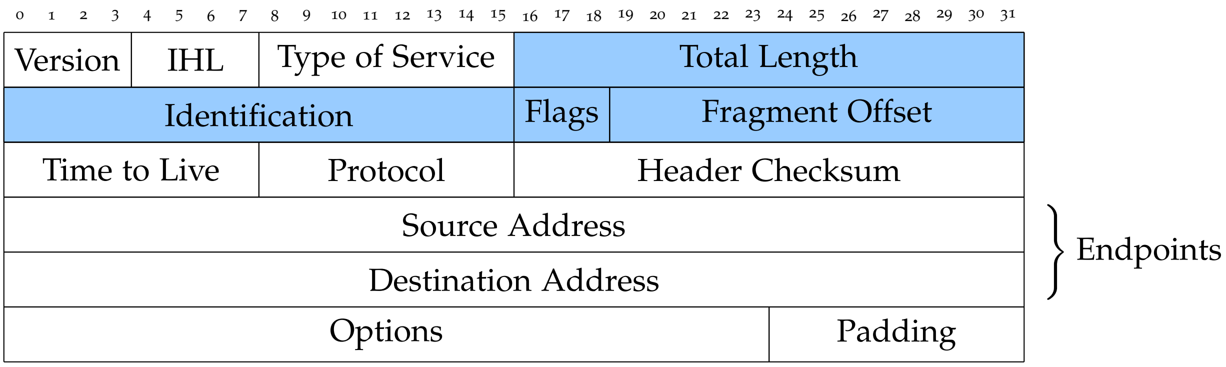


Figure 3.3.: Definition of the IPv4 Header. (RFC 760, 1981, p. 11) It contains information about the protocol version, length of header (IHL), the type of service the total length (header and data), identification of fragments that belong together, flags for signal that more frags are coming or an demand for don't use fragmentation and an offset of the payload.
🟦 : Used for fragmentation process.

stream or datagram, reliability and speed. It is a specification for the internet service quality. (RFC 760, 1981, p. 27)

The ***Total Length*** is the length in octets of the data and Internet Header. The minimum length that has to be accepted is 576 octets. The maximum possible length is 65535 octets. (RFC 760, 1981, 12ff)

The ***Identification*** is a value that is chosen by the sender to identify fragments that belong to the same data. This is the handle for assembling the fragments into the right packets even if the order at the receiver is different to the send order.

The ***Flags*** control the fragmentation:

0	1	2
0	DF	MF

Bit 0 is reserved and has to be zero. With the **DF**-Flag the sender tells the communication partner that this packet should not be fragmented on its further way. This can be useful if the receiver is not able to handle fragmented data. (RFC 760, 1981, p. 23) The **MF**-Flag indicates if more fragments are coming or if this is the last or only fragment. (RFC 760, 1981, p. 13)

The ***Fragmentation Offset*** indicates the position in the data where this fragment is located. The offset scale is divided in 64 Bits parts. So the payload size has to be a multiple of 8 Bytes. If not, padding has to be added.

The ***Time to Live*** field contains the remaining time that a packet stays valid for the net and is transported. The time is measured in seconds. If a communication member receives a packet it decreases the Time to Live field with the amount of time it took to process the packet. If there is no time measurement it is decreased with the minimum of 1.

IP is the routing protocol for many higher protocols. Which one is used is specified in the field ***Protocol***. TCP uses 6, Sprite-RPC uses 90. 253 and 254 can be used for experimental and testing purposes. (IANA, 2017[a])

> "[The ***Header Checksum***] field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero."
> RFC 791 (1981, p. 14)

It is used for error detection of the Header. There is no checksum for the payload. This has to be checked by the higher protocol. (RFC 791, 1981, p. 14)

The ***Source-*** and ***Destination Addresses*** are the IPv4 addresses for the endpoints. They are designed for routing packets in the whole world for over four billion endpoints and are separated in a net part and a local part for the device.

3.3.1. Fragmentation Procedure

The sender has to know the MTU, the maximum transmission unit, of the next network. If the MTU is smaller than the packet's Total Length, the packet is fragmented. The first fragment of this process has the Total Length of MTU and the second length is checked for size again. If the MTU is still too large, the packet is fragmented again. This is repeated until the last packet is equal to or smaller than the MTU. (RFC 760, 1981, p. 23)

The **header** of the first fragment is copied from the original packet. The MF-Bit is set, the Fragment Offset is set to 0 and the new Total Length is set to MTU. The Header Checksum is recomputed for the new header. For the remaining fragments the procedure is partly repeated but some options are not repeated after the first fragment. The ***IHL*** becomes smaller. The last fragment may have a smaller Total Length. (RFC 760, 1981, pp. 23,24)

3.3.2. Reassembling Procedure

During the resembling process it is necessary to find the right service access point where the header and payload info of each packet has to be entered. The key for this identification are the source, destination, protocol and identification fields of the IP header. Only when they are all the same they get assigned to the same endpoint. (RFC 760, 1981, p. 24)

If no packet with the appropriate endpoint has not been received yet, the endpoint resources are allocated. These resources contain a buffer for the data and header, a fragment block bit table, a value for the total length and a timer. The fragment block bit table marks which octet blocks already have been received. After transport was successful and the MF-bit is not set the Total Data Length is computed with the fragment block bit table. It also checked if all bits from 0 to Total Data Length are set. If not the receiver waits for the next fragment or until the timer expires. (RFC 760, 1981, pp. 24, 26)

The **timer** is initialized by the first fragment arriving. The timer can have a value between 1 and 255 seconds (4,25 minutes) if it is running and 0 if it is run out. During a running process a new fragment can increase the timer to a new value. So if the TTL-Value is higher the timer value assigns the TTL-Value. If the TTL-Value is smaller than the current TTL-Value the timer stays the same. If the timer runs out the endpoint resources are freed. (RFC 760, 1981, p. 25)

3.4. IPv6

IPv4 had some restrictions, security and addressing problems. It was released in 1981 where 4 billion possible endpoint addresses were out of reach. This claim was underestimated so a new version of the IP protocol has been designed: Version 6, which is displayed in Figure 3.4. In comparison to Version 4, which has a 32bit address space, IPv6 has an address space of 128bit. This will be sufficient for 2^{128} endpoints. The structure of the header is redesigned so that optional functionalities such as fragmentation are only included if needed. Fragmentation has its own header part that gets appended after the standard header and the other already included optional headers. The Next Header field indicates which header part comes next. (RFC 2460, 1998, p. 2)

The ***Version*** of the protocol is 6. The ***Traffic Class*** is used to determine different classes or priorities of packets. The ***Flow Label*** part signalizes realtime priority. (RFC 2460, 1998, pp. 2,25,26)

The ***Payload Length*** contains the number of bytes of the payload including the optional headers. The standard header of IPv6 has a fixed size of 80 bytes. The length of the whole packet is computed as $PayloadLength + 80$. In case of fragmentation there is no information about the whole length of data.

Term conflict: Payload

In the IPv6 definition (RFC 2460, 1998) the term **payload** includes the data from the upper level and the optional headers. To be consistent,in this thesis the term payload is used only for data from the upper level as described in Section 3.1.

The ***Next Header*** field indicates which block comes next. There can be additional IP-Headers or a specification of the upper protocol. The numbers are taken from RFC1700. So it is used in the same way as IPv4 uses its Protocol Field. The Fragment Header is defined as **Simple Internet Protocol Fragment** with the value 44. (RFC 1700, 1994, p. 9) (RFC 2460, 1998, p. 5)

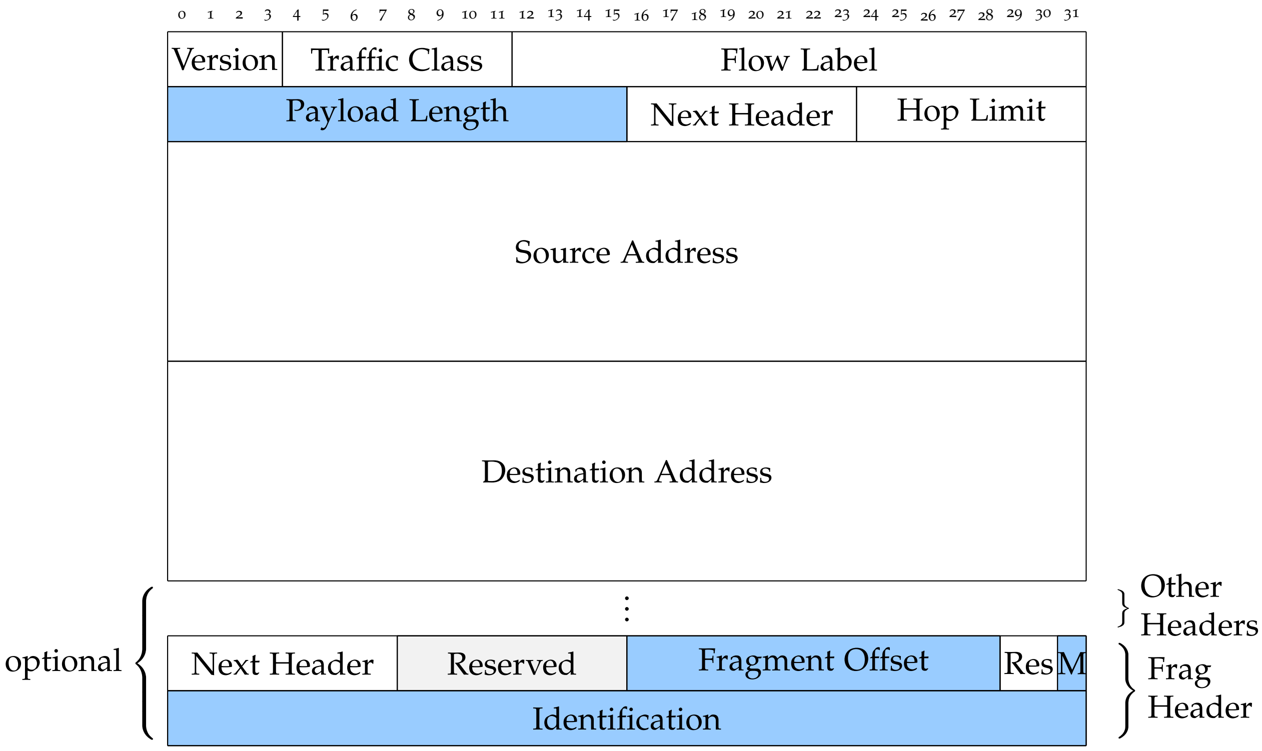


Figure 3.4.: Definition of the IPv6 header. It contains information about the protocol Version, priority/classes handling of routers, special handling of sequences, length of the payload, additional extra headers like fragmentation header and a Hop Limit for define a maximum amount of transport routers for one packet. The Source and Destination Addresses contain the IPv6 Addresses. All Headers are optional. The fragment header contains a description of the Fragmentation Offset, an extra flag for the indication if more fragments are coming and an identification field for finding the right endpoint. (RFC 2460, 1998, pp. 4 sqq.)
🟦 : Used for fragmentation process.

If the packet is forwarded by a node the ***Hop Limit*** is decremented by 1. This field is used to specify the amount of nodes this packet stays valid for it. It has a similar function as the Time To Live field in IPv4. The ***Source and Destination Addresses*** are the IPv6 addresses for the endpoints. They are designed for routing packets from one node to another. (RFC 2460, 1998, p. 5)

3.4.1. Fragment Header

For the fragmentation of an IPv6 packet an additional *Fragmentation Header* has to be included. Its structure is displayed in Figure 3.4.

The ***Next Header*** part assigns the next optional header or upper protocol, if it is the last one.

With the field ***Fragment Offset*** the sender tells the receiver in eight octet units the location of the payload of this packet in the completed data. This 13 bit value is sufficient for 65528 bytes of transport.

$Length_{max}(Data) = 2^{13} \cdot 4 = 65528 bytes$

The ***M - More-Fragment-Bit*** is set if this is not the last fragment. The ***Identification*** field is used to reassemble the right fragment in the appropriate buffer. Together with the source and destination address, the identification forms the so-called Service Access Point.

3.4.2. Fragmentation Process

Before the fragmentation of the original packet can take place it has to be divided into a fragmentable part and an unfragmentable part. The unfragmentable part has to be part of every packet because it is needed by every node to forward and process the packet, e.g. the routing header. The headers which are not fragmentable are placed before the fragmentable header. The headers that are fragmentable are placed after the fragment header. (RFC 2460, 1998, pp. 19,20)

The data and the fragmentable header become the fragmented data. This data is divided into fragments so that the unfragmentable headers, the fragment header and the fragments fit into one packet that is equal to the MTU of the net. This also includes paths between inner nodes because they are not allowed to be fragmented if a subnet has a smaller MTU. (RFC 2460, 1998, pp. 18,19,20)

3.4.3. Reassembling

As mentioned above the Service Access Point is defined through the source- and destination addresses and the Fragment Identification. Traffic Class and Flow Label are ignored for that. The reassemble starts when all fragments have been received. There is a timer that starts with the reception of the chronologically first fragment and stops waiting for new fragments after 60 seconds. If this occurs the fragments are dropped and allocated resources are freed. (RFC 2460, 1998, p. 22)

If the reception of all fragments is successful, the unfragmented part of the header from the first fragment is the basis for the original header. The first fragment s identified by $FragmentOffset = 0$. The *Next Header* field from the last fragment is used for the original fragment. The original payload length is computed through the unfragmentable part of the header from the first fragment and the total length of data which is computed from the fragmentation offset of the last fragment and the payload of the last fragment. (RFC 2460, 1998, p. 21)

3.5. IKEv2

The Internet Key Exchange Protocol Version 2 conditionally supports fragmentation. Its underlaying protocol is UDP which uses ports 500 or 4500. (RFC 7296, 2014, p. 72) Internet Key Exchange is used in the IPSec Protocol suite to open a secure tunnel for communication. Two nodes open a communication channel that is calld IKE Security Association. IKE only communicates with call and response. The IKEv2 header is displayed in Figure 3.5.

The ***Security Parameter Index (SPI)*** of ***Indicators*** and ***Responder*** are used to open and use a Security Association connection with a Diffie-Hellman exchange to have a common secret. (RFC 7296, 2014, pp. 10,73) The ***Next Palyoad*** field indicates which kind of payload comes next. In IKE it is possible to append different kind of payload after each other. If there is an encrypted payload appended, it has to be the last one. (RFC 7296, 2014, pp. 10,110) The ***Major Version*** indicates the highest supported version of the protocol. Version 2 supports fragmentation. There is no special version that indicate if the end supports fragmentation. The ***Minor Version*** indicates the minor protocol version that is still in use. Since there is no comparability this should be set to 0. The ***Exchange Type*** specifies types of transportation which indicate orders of payloads. ***Flags*** indicates if this packet is part of a call or a response. The ***Message ID*** is used for identifying a packet. IKE is a reliable protocol and supports retransmission through a timer event. This helps to prevent replay attacks. The ***Length*** is the length of the packet. (RFC 7296, 2014, pp. 24,25,73,74,75)

In the original description of IKEv2 there is no specification for fragmentation. There is only a reference to IP fragmentation that can be used in the lower layer. (RFC 7296, 2014, p. 24) But there is an extra specification which describes fragmentation of the encrypted data. The main goal of this is to implement fragmentation in IKEv2 in order to avoid lower layer IP fragmentation since there are some routers that can not handle it and there are security concerns and attack scenarios for IP fragmentation. Fragmentation of IKEv2 can only be used if both communication partners support, use and expect this extension. There is no field that indicates that the fragmentation modification of the IKEv2 protocol is used. (RFC 7383, 2014, pp. 1,2)

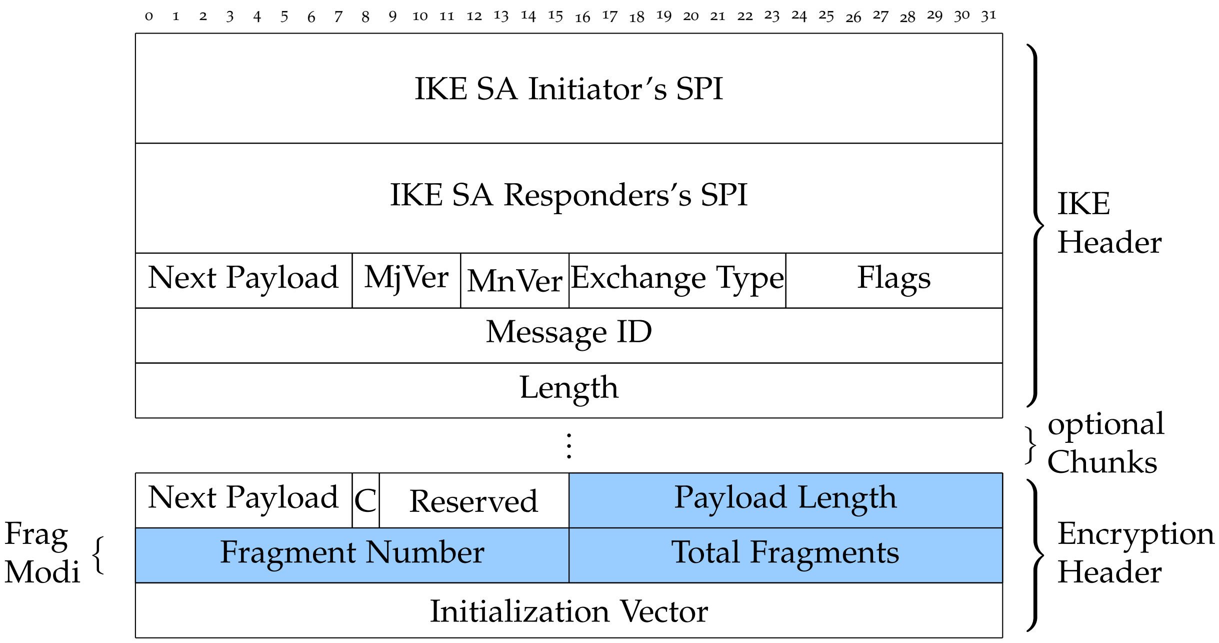


Figure 3.5.: Definition of the IKEv2 header modified encryption for fragmentation. It contains an initiator and responder security token for establishing a secure connection, a specification of the next type of payload, the minimum and maximum supported versions, exchange type for payload, flags that indicates the role of the sender, Message ID for retransmission and the total Length of the packet. Then optional headers are appended. A modification of the encrypted header may use fragmentation. It also contains a next payload field, length of the payload, an fragment number for every fragment, the total number of fragments and an IV that has been used for the encryption of the payload. (RFC 7296, 2014, pp. 71 sqq.,111)
🟦 : Used for fragmentation process.

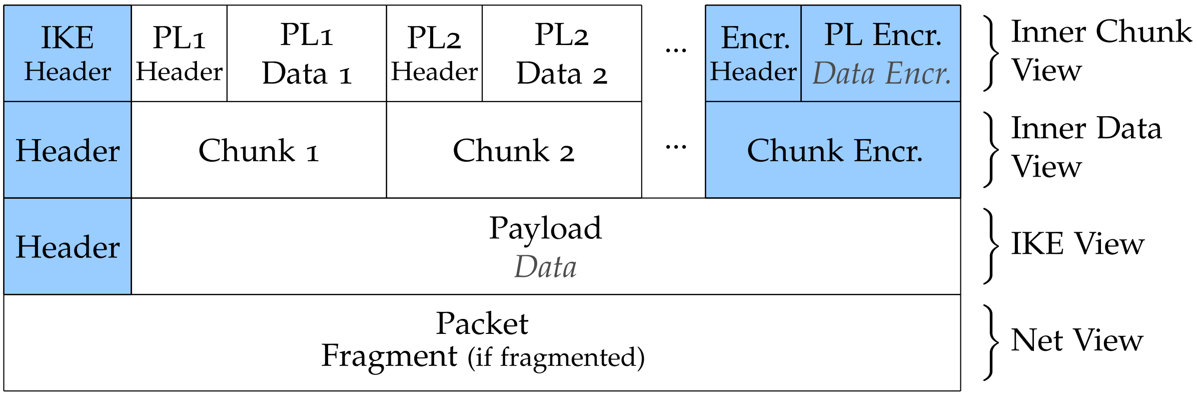


Figure 3.6.: Structure of IKEv2 packet, which starts with the mandatory IKE Header and is followed by at least one chunk (Inner Data View). These chunks are appended after one another. Each chunk consists of a header and an inner payload, which simultaneously is the inner data. The last payload can be encrypted.
The encrypted payload is fragmentable. If it is fragmented the payload does not consist of the whole inner Data.
🟦 : Used for fragmentation process.

Term conflict: Payload

In the Definition (RFC 7296, 2014) the term **Payload** is used like it is used in the *Inner Data View* in Figure 3.6. To be consistent and precise the term payload depends on the layer that is described in this thesis. The default context is the *Inner Chunk View*. If used differently it is mentioned.

The encrypted payload has to be sent last as shown in in Figure 3.6. It can also be the only payload in the whole packet. The ***Encrypted Header*** starts with the **Next Payload** field that indicates the type of the next payload. If the encrypted payload is fragmented the first fragment contains a number that specifies the data that is currently encrypted. The next fragments must be zero. (RFC 7383, 2014, p. 7) (RFC 7296, 2014, p. 111)

The ***Payload Length*** contains the length of the Encryption Header and payload of this fragment. There is no declaration of the whole encrypted data length. (RFC 7383, 2014, p. 7)

The ***Fragment Number*** and **Total Fragments** fields are only coded in the modified version of the protocol. In the original version these values form the ***Initialization Vector*** as a 64 bit value. With fragmentation its size is 32 bits. The Fragment Number identifies the fragment. It starts with 1 and gets incremented with the next fragment that is sent. The Total Fragments field is the number of fragments that contain the encrypted data. (RFC 7383, 2014, p. 7)

3.5.1. Fragmentation Process

As basis for the fragmentation the MTU of the network should be used. If there is no information about it IKEv2 uses the default value of 576 bytes for IPv4 and 1280 Bytes for IPv6. The first fragment may contain one or more unencrypted payloads before. The following has to be true so that fragmentation is possible:

$$Length(Header_{IKECommon} + Payload_{unecrypted} + Header_{Encryption}) \leq MTU$$

Otherwise the encrypted part has to be sent in a fresh packet with a new ***Message ID***. (RFC 7383, 2014, pp. 8,9)

The first fragment has the ***Fragment Number*** 1 and the ***Next Payload*** field contains a number that specifies the encrypted data. The total fragments has to be computed before the first fragment is sent due to the total fragments value field that has to set in every fragment. All packets except for the last one have the length of the MTU. (RFC 7383, 2014, p. 11)

The next fragments contain only one payload which is encrypted. The correct fragment number is inserted but the Message ID has to stay the same. (RFC 7383, 2014, p. 11)

3.5.2. Reassembling

The receiver detects a fragment if the packet contains an encrypted header where the Total Fragments field is 2 or more and the Fragment Number field is between 1 and the Total Fragments field. There are contradictory descriptions on how to handle a changed Total Fragments value. In case of already started fragmentation, one requirement is that the Total Fragments value has to be greater or equal, to the number of fragments that has already been received. Another requirement of a still running reassembling process is a greater Total Fragment field demands that all already received fragments have to be dropped and only the current fragment has to be stored to wait for more fragments to arrive. (RFC 7383, 2014, p. 12)

A packet is only accepted if it is not a retransmission. To be more precise, if the ***Message ID***, the ***Fragment Number*** and the ***Total Fragments*** fields have already been received in this combination the packet is dropped. There is also an internal integretiy check field in the encrypted payload. If it fails all packets are dropped. (RFC 7383, 2014, p. 12)

3.6. SCTP

The Stream Control Transmission Protocol (SCTP) is a transport protocol on top of IP. It supports similar functionalities as TCP for a communication between two endpoints. Supported features are full-duplex transmission of data (includes a retransmission mechanism) and flow control. (RFC 3286, 2002, p. 1)

It also supports features like multi-streaming transmission where two transmissions between two endpoints can be handled at the same time. If packets of one stream are lost it does not affect the other streams. (RFC 3286, 2002, p. 2)

SCTP supports more than just the transport of bytes from one endpoint to another. It transports data chunks from one part to another. There is one common header and chunks that are appended. Every chunk may have a different type of content. Fragmentation of chunks may happen with the data chunk. The header structure is displayed in Figure 3.7.

The underlying protocol is IP. The combination of IP addresses and ***Source-*** and ***Destination Port Number***s are the endpoints for the communication. The ***Verification Tag*** identifies the other communication partner. The receiver checks if it is still the same value that is given to the sender during

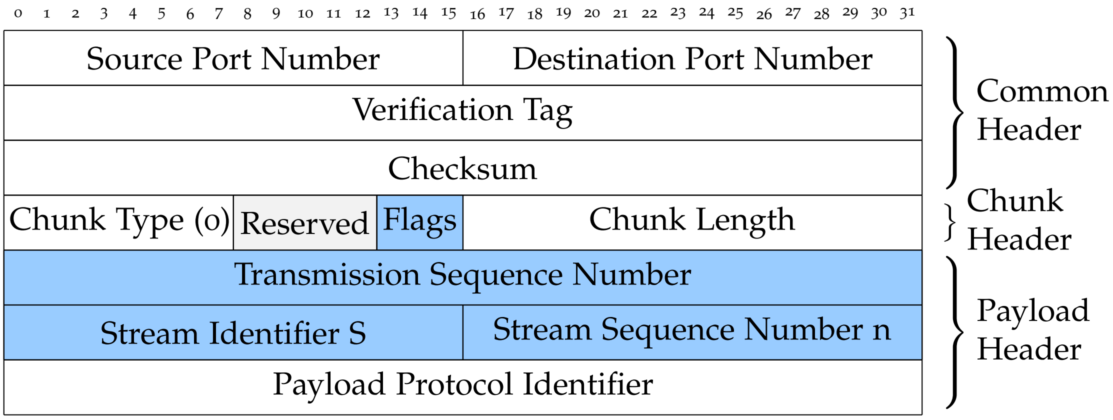


Figure 3.7.: Structure of an SCTP Packet with the chunk and payload headers. The headers consist of the port numbers of the source and destination endpoints, a verification tag to identify the sender, a packet checksum, flags for classifying data and fragment information and a length of the whole chunk. The Transmission Sequence Number identifies the sent payload chunk, the linked stream and the stream sequence number and an upper protocol identifier. (RFC 4960, 2007, pp. 22, 23)
🟦 : Used for fragmentation process.

the association phase. In contrast to that the sender uses the value that is known from the receiver. There are exceptions for the first packet, shutdown and abort of the communication. The ***Checksum*** field uses the CRCD32c algorithm to calculate a checksum for the whole packet. (RFC 4960, 2007, pp. 6,7)

The chunk itself contains of a chunk header. It starts with a ***Chunk Type***. There are extra chunk types for Payload Data, as used in Figure 3.7, various types for establishing an SCTP connection, aborting or shutdown of the connection. For the Payload Data the value is 0. (RFC 4960, 2007, pp. 17,18)

The ***Chunk Flags*** controls the Fragmentation:

0	1	2
U	B	E

The **Unorderd Bit U** indicates that there is no ***Stream Sequence Number*** in this chunk and so the relative position does not matter. In this case it is set to 1. If the order is important this bit shall be 0. The **Beginning Fragment B** indicates that this is the first fragment. The **Ending Fragment E** indicates that this is the last fragment. If no fragmentation is used **B** and **E** has to be 1. If both are 0 it is a fragment in between. (RFC 4960, 2007, pp. 22,23)

The ***Chunk Length*** indicates the length of the chunk header and payload.

The ***Transmission Sequence Number (TSN)*** is incremented with every data chunk that is transported. Every fragment has its own TSN value. (RFC 4960, 2007, p. 23)

The ***Stream Identifier S*** and ***Stream Sequence Number n*** are used for identifying and handling different streams. Each stream Identification field represents another layer of endpoints besides the ones defined in Source and Destination Port. The Stream Sequence Number sorts the data. In case of fragmentation the Stream Sequence Number field does not change. (RFC 4960, 2007, pp. 23, 24)

The ***Payload Protocol Identifier*** is not used by SCTP. It can be used by the upper protocol to identify the payload that is transported. It has to be set in every fragment to ensure that every node in between can access this information. (RFC 4960, 2007, p. 24)

3.6.1. Fragmentation and Reassembly Procedure

A data chunk is fragmented when all data chunks plus the common header are larger than the MTU. In that case the last data chunk is fragmented. To indicate that fragmentation is used the **Beginning Fragment** Flag is set. The next fragment size depends on the missing part of the data that has to be transmitted. If the common header, the data chunk header and the missing data fit into in one fragment the **Last Fragment** Bit has to be set. (RFC 4960, 2007, p. 91)

For the next fragments the ***Common Header***, the ***Chunk Header*** and the ***Payload Header*** are copied from the first one. The TSN is incremented. The ***Stream Sequence Number*** stays the same. To identify that this fragment belongs to the same data. The ***Chunk Length*** and the ***Checksum*** have to be adapted. If it is the last fragment the **Last Fragment** bit has to be 1. (RFC 4960, 2007, p. 91)

A receiver checks if a data chunk is fragmented by looking at the ***Flags***. The first fragment is indicated with the ***Flags*** (B:1, E:0). The chunks are queued for reassembling. The receiver waits for more fragments with the same Stream Identifier and the next Stream Sequence Number to come. The chunks are sorted by TSN. If the buffer is filled completely before all fragments could be received SCTP supports the feature of partial delivery of already received packets to the upper layer. (RFC 4960, 2007, pp. 91,92)

3.6.2. (Selective) Acknowledgment Mechanism

SCTP supports Selective Acknowledgment (SACK) of data chunks with a special chunk type. The Selective Acknowledgment chunk perform acknowledgment of fragments with no errors in transmission but it also provides information about chunk gaps and duplicate transmission of chunks to the sender. (RFC 4960, 2007, pp. 34,35)

The ***Cumulative TSN Ack*** field is the basis of the information of gaps and duplicated fragments. It contains the TSN value of the last chunk that has been received before a gap. The ***Advertised Receiver Window Credit (ARWC)*** redefines the buffer space in bytes. The ***Number Gaps*** and ***Number***

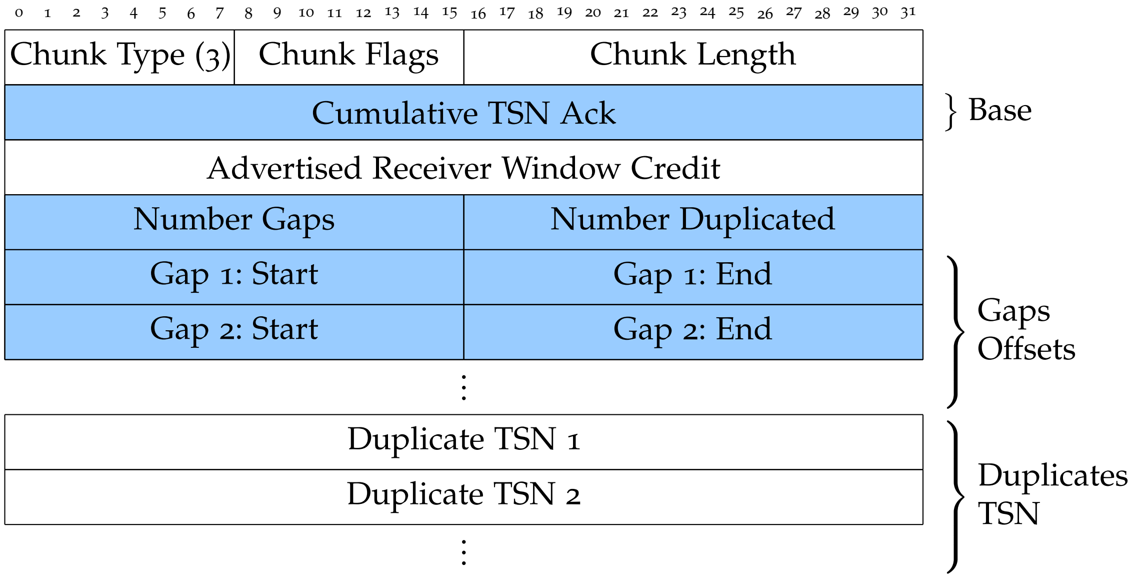


Figure 3.8.: Structure of an SCTP Selective Acknowledgment Chunk. It provides Information about TSN gaps that have not been received yet and about chunks with the same TSN which have been received multiple times. (RFC 4960, 2007, p. 35)
🟦 : ACK Information inclusive gaps.

Duplicated indicate how many gaps and duplicates there are. This indicates which fields are used afterwards. (RFC 4960, 2007, pp. 35,36)

Each gap has a start and an end ***TSN***. The ***Start*** field indicates the offset to the last received ***TSN*** before the gap to the ***Cumulative TSN Ack***. The ***Stop*** field indicates the offset to the last missing TSN number in the gap from the ***Cumulative TSN Ack***. These two fields are needed for every gap. (RFC 4960, 2007, p. 36)

The ***Duplicate TSN*** fields contain the TSN numbers of duplicate ***TSN*** numbers received since the last SACK has been sent. If there is more than one duplicate TSN each gets its own entry. (RFC 4960, 2007, p. 37)

3.7. RPC

The Remote Procedure Call protocol is used for interprocess communication that is normally based on a request response model where the client makes a request, the server processes the request and sends back a response. RPC itself is no defined specification of a protocol. There are several implementations that are not compatible with each other.

PROFINET uses the CAE Spefication of RPC, which this thesis is based on. (CAE Specification, 1997, p. 578) The common RPC header is described in Figure 3.9. There are different kinds of packets that have different functionalities and services which are called Protocol Description Units (PDUs). Each PDU may consist of three different parts, a header, a body and an authentication verifier. There are 20 different PDUs described. (CAE Specification, 1997, p. 575)

The ***RPC Version*** represents the version of the protocol. For the the described specification it is version 4. The ***Packet Type*** contains the PDU Type that is used by this packet.

The ***Flags*** are divided into two segments. The first contains the fragmentation information:

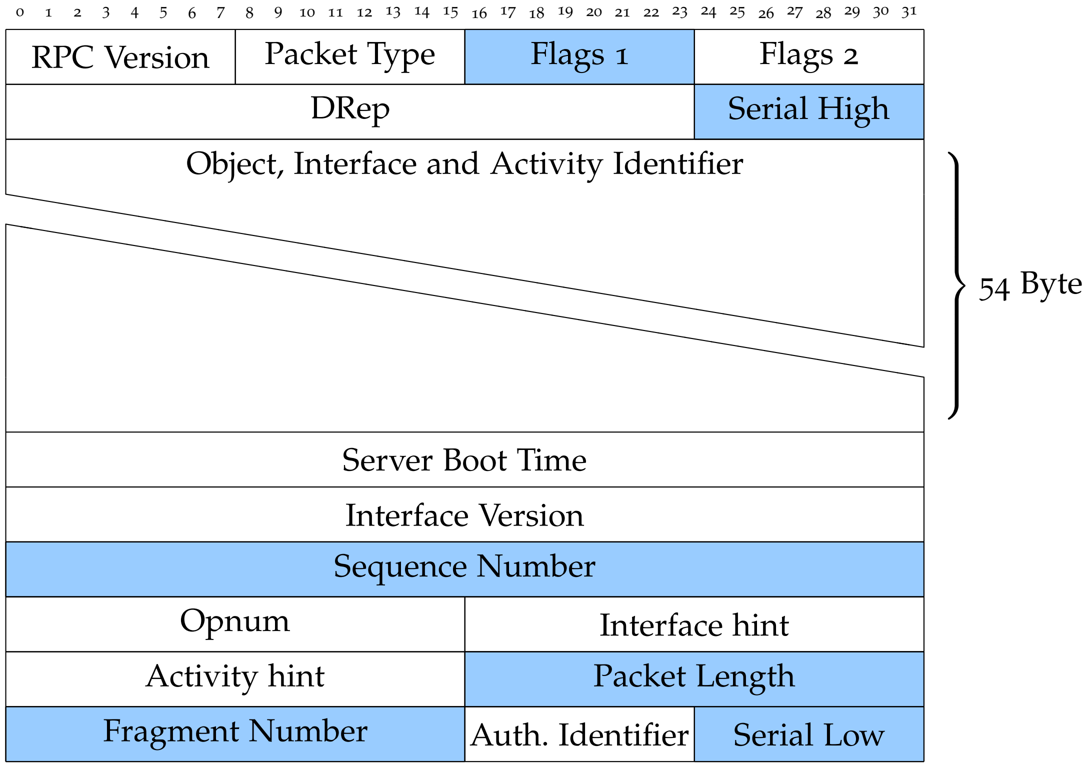


Figure 3.9.: Connectionless RPC header. It contains information about the supported RPC version, the packet type, control flags and data decoding information. It contains Identifiers from the object, interface and activity, which uses 54 Bytes, the interface version, a unique sequence number for each call, a fragment number for each fragment of one call and an identifier that specifies the protocol that is used for authentication of the body. (CAE Specification, 1997, p. 578)
🟦 : Used for fragmentation process.

0	1	2	3	4	5	6	7
RES	Last F	Frag	No Fack	Maybe	Idem.	Broad	RES

🟦 : Used for fragmentation process.

The **Last Frag** Flag indicates that this is the last fragment of a fragmented transmission. The **No Fack Flag** demands of the receiver of this fragment not to send an acknowledgment PDU for this fragment. Client and Server may use this.

The **Maybe idempotent** Flag indicates the classification of a request. There is no guarantee that a Maybe Request is delivered. No Ack will be sent and in case of a failure no retransmission is executed. If the request is labeled as idempotent the transmission is guaranteed. After a timeout, the sender executes a call again. This call may be executed more than once. If the request should only be executed once because a second execution would bring a different result the request is labeled At-Most-Once. One and the same call is only executed once if the receiver detects it is a retransmission. To do that the Maybe and Idempotent Flag have to be Zero. The Broadcast is a request to all hosts in the network. (CAE Specification, 1997, pp. 586, 587)

The ***DRep*** field describes the encoding of the packet for chars, integers and floats that has been used. Encodings like LSB or MSB, unsigned or signed char or floating point or fixed floats are described here.

The ***Serial High*** and ***Serial Low*** field are the higher and the lower byte of the 16bit sized serial number that identifies fragments of one transmission. The ***Fragment Number*** identifies a fragment during one call. In case of retransmission the number is also incremented. The serial number is only useful for request and response PDUs. For other PDUs this field is zero. (CAE Specification, 1997, pp. 580, 581)

The ***Object-*** , ***Interface-*** and ***Activity Identifiers*** together are the endpoint to where the call shall be delivered. One object may have more interfaces and one interface may have more activities. Every identifier consist of 16 bytes, so the endpoint identifier consist of 48 bytes. The ***Server Boot Time*** contains the time that the server needs to get started and is designed to detect if the server has crashed. The ***Interface Version*** allows to specify different versions. (CAE Specification, 1997, p. 581)

The ***Sequence Number*** identifies a call. Each fragment of one call has the same sequence number. In case of a retransmission the sequence number does not change. With every new call the number is increased. In case of a request or response the PDU has the same Sequence Number. (CAE Specification, 1997, p. 582)

The ***Opnum*** identifies a particular operation. The ***Interface and Activity hint*** fields can be used by the implementation for optimization of lookups or other information. (CAE Specification, 1997, p. 582)

The ***Payload Length*** is the length of the payload. The maximum is 65528 bytes. In case of fragmentation this is only the length of one fragment. (CAE Specification, 1997, p. 582)

The ***Fragment Number*** identifies the fragment of a fragmented PDU. In case of a request or response PDU the first fragment has the value 0. With every new fragment that is sent the value is incremented. In case of a retransmission the value does not change. If the PDU type is a FACK the value is set to the last successful fragment transmission without gaps in the fragment transmission.

The ***Authentication Protocol identifier*** defines the used authentication protocol. For this version it is only possible to use no encryption or the OSF DCE Private Key Authentication. (CAE Specification, 1997, p. 583)

3.7.1. Fragmentation Process

If the MTU of the net is smaller than the request or response payload plus RPC header plus optional ***Authentication Identifier*** then the packet has to be fragmented. Only request and response packets vary in their length, so they are the only ones that may be fragmented. In order to signal fragmentation the fragmentation flag has to be set. With the No Fack flag the sender decides that no acknowledgment of this fragment is wanted. For the very first call after connection establishment it is recommended to use a window size that the receiver is sure to support. The ***Serial Number*** and ***Fragment Number*** will be set to 0 for the first fragment. The packet length is only the length of the current fragment. (CAE Specification, 1997, p. 585)

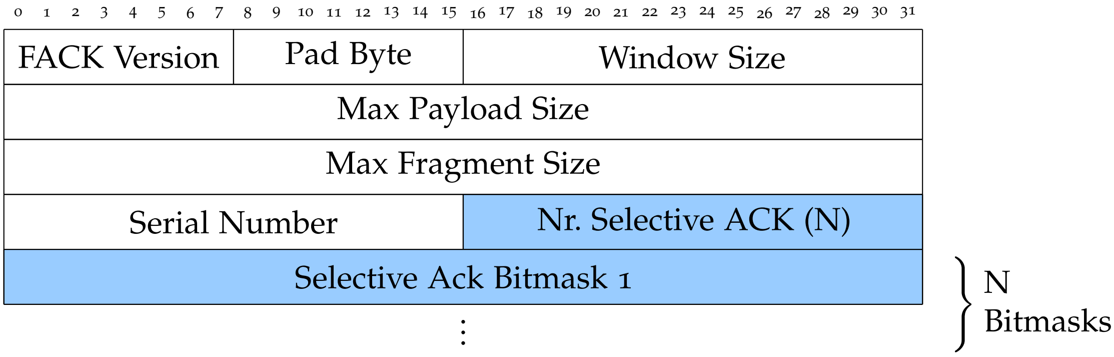


Figure 3.10.: This is the optional Fragment-ACK payload(CAE Specification, 1997, p. 578). It contains information about the Fragment ACK version and a Window Size which indicates how much payload in KB the receiver is able to process. The Max Payload Size includes the maximum payload for the whole request or response (includes all fragments). The Max Fragment Size is the maximum payload of one fragment. The Serial Number is the serial number of the sender that triggered this FACK. The Nr of Selective ACK is the amount of bitmasks for the Selective ACK which allows the receiver of the ACK to retransmit only the gap fragments. (CAE Specification, 1997, pp. 584, 585)
🟦 : Selective ACK information.

If the sender demands an acknowledgment the receiver checks the serial, fragment and sequence number and sends a FACK PDU. The FACK PDUs ***Fragment Number*** is the same as the highest fragment number that has been received without a gap. The FACK PDU may contain a body like in Figure 3.10 but it does not have to. For acknowledgment it is not necessary but it supports some extra features like window size, MTU adaption and selective acknowledgment. The Nr of Selective Ack bitmasks are described in ***Nr Selective ACK (N)***. The first bitmask shows the selective acknowledgment for the first 32 fragments after the fragment number. The second bitmask would show the SACK from 33 to 64 fragment numbers after selective acknowledgment. The formula for the computation of the selective acknowledgment is: (CAE Specification, 1997, p. 583)

$$fragmentNumber + (numberOfBitmask) * 32 + indexInBitmask$$

The sender of the request or response PDU waits for an incoming FACK PDU and retransmits fragments if necessary. If the last fragment is sent the Last Fragment Bit header is set. The behavior of the server after the transmission of a request is completed depends on the request classification. If the class is At-Most-Once then an ACK PDU is sent. The ACK PDU has no body data. (CAE Specification, 1997, p. 583)

By looking through these protocols a comparison of the fragmentation strategy follows.

3.8. Comparison

What the protocols have in common is that they support fragmentation. But they have different users, different scopes of how many features are supported and different hierarchies where they are placed in protocol stacks. The OSI model classifies network layers in seven main layers, which begin on the one end at the Physical layer, which represents the actual transported bits, and ends at the application that is used. The Layers DataLink and Session Layer are in between, as represented as Borders in Table 3.1. The Data Link layer transports frames without errors to another node. The Network Layer routes packets through networks over nodes. The Transport Layer guarantees the transport of a bytestream and/or messages between nodes. The Session Layer implements process orientated connections that are able to handle timeouts without a complete reset. The table gives an overview of how the protocols can be assigned to elements in a modified OSI model. This model separates the transport layer in two layers: byte orientated and message orientated.

The **Real Time Acyclic protocol (RTA)** transports alarms (a kind of message) over the net. There is no routing over IP. The alarms are addressed directly through MAC addresses and are checked and forwarded to the upper layer. RTAs transportation mechanism is similar to TCP's. They both support an ordered, reliable, full dublex, end-to-end connection between nodes. The main difference is that it separates acknowledgment and data transportation with different PDU types. RTA transports messages and not a bytestream.

As mentioned before, **TCP** is the well established and a well known protocol for transporting bytestreams. The typical setting of TCP is in combination with **IPv4** or **IPv6** as the network layer below it. IP does not necessarily

Table 3.1.: Modified OSI model and assigned protocols. The compared protocols and the original RTA protocol are marked regarding their layer in the OSI Model. The Transport Layout is split up into message oriented transportation and byte oriented transportation, which is not an official OSI specification. RTA transports alarm messages without IP layer. TCP realizes a byte stream. IPv4 and IPv6 route packets through large nets. IKEv2 uses UDP as transport protocol and establishes an authentic and confidential connection. SCTP performs a message orientated connection on top of IP. RPC may sit on top of TCP or UDP. It supports session handling through the request response mechanism. (CCITT, 1988)

* RTA addresses nodes through MAC addresses and does not use IP. But an extension with IP is possible.

Layer	RTA	TCP	IPv4	IPv6	IKEv2	SCTP	RPC
Application							
Presentation							
Session					🟦		🟦
Transport Messages	🟦				🟦	🟦	🟦
Transport Byte	🟦	🟦				🟦	
Network	*		🟦	🟦			
Data Link							
Physical							

have to be used as network protocol below TCP. But there has to be another information source about the segment size. (RFC 1791, 1999)

IP establishes the identification of nodes and the transportation of packets through the net. These packets are not reliable and may be jumbled.

IKEv2 is a higher abstracted protocol. It transports messages over a UDP bytestream to set up a security association. In respect of this association this protocol establishes a session through generating a valid shared secret that makes a confidential and authenticated connection possible.

The **SCTP** protocol is similar to TCP but has some extra features and differences. It transports whole messages that are if necessary fragmented and internally reassembled. It is possible to have multiple message streams in parallel through the same connection.

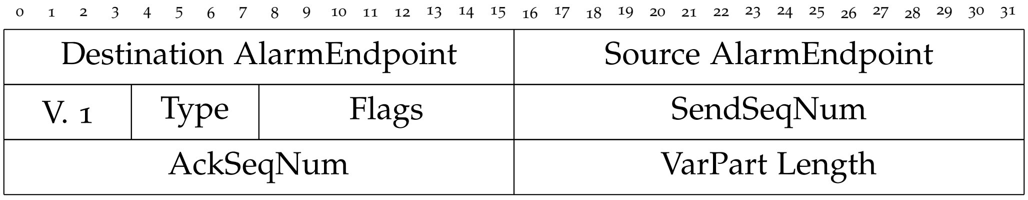
RPC is a typical protocol for the session layer. The two roles client and server interact through access points and sessions. The client makes a request and waits for a response from the server. TCP or UDP is used as transport protocol.

The protocols are compared by category. The packet, especially header structure, is the beginning.

3.8.1. Packet Structure

The packet structure describes the structure of the protocol in the network. To be more precise this chapter analyzes the structure of how the fragmentation information is inserted and how much information is needed for a standardized and representative fragmentation process. The implementation of fragmentation has to be adapted for RTA. The header of RTA is displayed in Figure 3.11.

The fragmentation fields of **TCP** and **IPv4** are present in every common header and so in every packet that is sent. Due to the functionality of TCP as bytestream transport it is necessary to include the information in each packet. It is the key feature of TCP to implement a reliable full dublex



Destination AlarmEndpoint			Source AlarmEndpoint
V. 1	Type	Flags	SendSeqNum
AckSeqNum			VarPart Length

Figure 3.11.: RTA Header of Version 1.

connection. In IPv4 fragmentation is optional. But even if fragmentation is not used the fields are present in every packet.

This has changed in **IPv6**. IPv6 uses a common header and includes a fragmentation header if necessary. So if no fragmentation is used 8 bytes of information do not have to be included in the packet header.

The addition of optional fragmentation headers is also used by the **IKEv2** and **SCTP** protocols. Both use data chunks for transportation. That means that the payload of one packet consist of queued inner packets, called chunks. These chunks consist of their own headers and payload. The structure is displayed in Figure 3.6 on Page 22. It allows to include fragmentation information per chunk. The common header does not contain fragmentation fields but in IKEv2 every encrypt chunk and in SCPT every data chunk contains fragmentation information. Only these chunks are fragmentable.

The fragmentation fields of **RPC** are in the common header and are repeated in every RPC packet. Depending on the PDU extra information for fragmentation is possible. As mentioned before, only the request or response PDUs will become larger than the MTU because they transport payload from upper layers. There is no extra PDU for fragmented PDUs.

The used fragmentation bits per protocol are shown in Figure 3.12. The protocols differ in their scope and functionality. In order to make them comparable the needed fields for fragmentation are adapted to transport 64 Kbytes to support the maximum ***VarPartLength*** of RTA which is 1432 bits and to support 255 unique packets that have to be put in order. This amount of ordered unique packets guarantee the correct transport of the maximum data size of 64 Kbytes.

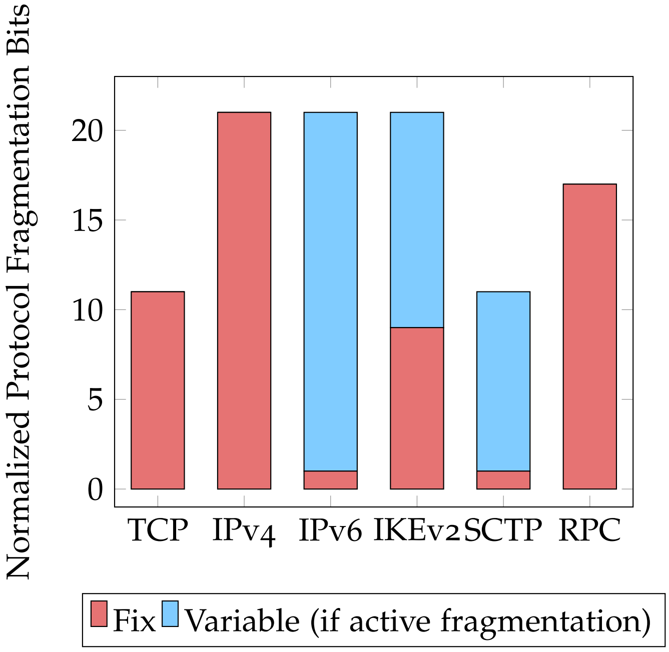


Figure 3.12.: Comparison of used fragmentation bits per packet. The needed bits are normalized to transport 64KBytes of data, 1400 maximum payload per packet/fragment and order clarity for 255 packets. The red bar symbolizes the fixed part of fragmentation bits which is not reducible in case of no fragmentation, and is the minimum. The blue bar represents the bits that are needed in case fragmentation is used and is the maximum, broken down in Appendix G.

Every protocol needs to know how many bytes the payload consists of. This information is not included in this comparison even though it would also be mandatory if the protocol would not support fragmentation.

TCP, IPv4 and RPC have fixed fields for fragmentation that do not vary if fragmentation is not used. In contrast, IPv6 has optional headers that can be appended. As mentioned before, IKEv2 and SCTP are protocols that are based on data chunks which are build up through chunk headers and chunk payload. The chunk header often contains all or a part of the fragmentation information. To signalize that there is an extra fragmentation header or chunk it needs an extra bit. In the compared fragmentation strategies eight bits are used. This is the typical internet standard of how to describe the next content in the protocol that is coming.

All protocols are quite similar in using unique identifiers for their packets. The name differs. In TCP it is the SequenceNumber, in IPv4 and IPv6 it is called Identification, in IKEv2 it is the ***Message ID***, in SCTP it is the ***Transmit Sequence Number*** and in RPC it is the ***Serial Number***. In IPv6 the field is in the optional header but in IKEv2 the field is in the common header so it is there even if it is not used. The ***Message ID*** is not only used for fragmentation and is used by other services of the protocol as well. This is the reason why IKEv2 has a fix fragmentation using 9 bytes.

The Offset Representation has two main ways of realization. IPv4 and IPv6 use byte offsets. Their offset field shows the current position of the payload in the receiving buffer with multiples of 64 bytes. An offset of 10 means that this payload has to be inserted 640 bytes from the beginning. This allows the retransmission of some parts and interleaving. The data has to be padded if the length isn't an multiple of 64 bytes. For a data size of 64 Kbyte 11 bytes are necessary. The maximum value is $0x400$.

The other way of matching the place in the buffer and the part in the fragment is the Fragment Identification. This is coded in the ***Fragment Number*** (IKEv2 and RCP), ***Transmission Sequence Number*** (SCTP and TCP). The first fragment is inserted at the beginning of the receiving buffer. The second fragment is appended afterwards. It depends on the previous fragment. This kind of fragmentation makes an ordered transmission or a well known size of the payload mandatory.

The **Flags** differ. IKEv2 has no flags at all. All the others have a flag that symbolizes that more fragments are coming. If this flag isn't set this is the last Fragment. In IKEv2 this information is not coded in the ***Total Fragments*** field which contains an information on the number of fragments. If the fragment number and the total fragment number is the same the receiver knows that no more fragments will come. IPv4 has a **DO FRAG**, RPC an FRAG and SCTP a **FIRST FRAGMENT** flag. All symbolizes that Fragmentation is used. RPC has a **NO FACK** flag that symbolizes that no acknowledgment of the fragment is desired and TCP has an **ACK** flag that means the opposite.

IPv4 and **IPv6** have 21 bits of memory because they do not support ordered transmission. So this information can not be included in ***Sequence Numbers***. In IKEv2 the ***Message ID*** helps also to prevent replay attacks.

Another important category for this comparison is acknowledgment of fragments. Is there any acknowledgment at all and if yes how is it done?

Acknowledgment

RTA itself supports acknowledgment through its ACK Number. The **TCP** and **RPC** acknowledgment number has the same size and works similarly. **IPv4**, **IPv6** and **IKEv2** do not have a functionality to acknowledge packets. So they are no reliable protocols. **SCTP** has its own chunk type for selective acknowledgment, shown in Figure 3.8. This chunk has an ACK number and optionally provides information of gaps and information about doubled sent packets.

Selective Acknowledgment is supported by SCTP and RPC but their implementation is different. SCTP tells how many gaps are described and shows the relative start and end position of the gap. In a scenario with few gaps which consist of a series of fragments that could not be received this implementation would be very useful. Since there is only a maximum WindowSize of 15 as a requirement for RSI there could not be that many gaps. So in the worst case that every second fragment is missing and the maximum window size is used the SCTP mechanism would use seven gaps are possible. The Selective Mechanism needs $3 + 6 * n$ bits. 3 bits for maximum 7 numbers and 2*3 bits for marks of start and end of gaps. RPC uses a different mechanism. It uses Bitmasks for every sequence number of a fragment that is missing. There is a field that describes the number of bitmasks. RPC now uses a maximum WindowSize of 2. RSI will also have the default value 2 but is extendable to eight packets. So for the upcoming scenarios seven bits will be necessary. This solution uses always fewer bits for selective acknowledgment with the exception of SACK. In this case it is two bit smaller.

A deeper description of acknowledgement realizations other parts of the communication are described in the next section about sequence diagrams.

3.8.2. Sequences

The most important points for this comparison are the sequences that show how the protocols are implemented and how the fragmentation is included into the process of communication. The Remote Procedure Calls shall use RTA as transport protocol. The current PROFINET RPC sequence diagram is displayed in Figure 3.13.

PROFINET RPC is a modification of the original connection less RPC protocol that has added additional opcodes like **PrmEnd**, Cancel or Control. The sequence shows the communication between the controller and the device. Each of them is controlled by a user which represents the upper application Layer. The user triggers events like DeviceAdd, ApplReady and ARCancel. These events are displayed in this figure because they may cause cross over calls. Each controller/device instance is addressed over the net through an AccessPoint **AP**.

During the **Connection phase** the controller sends a connect request from an AP **A** to the device Endpoint Mapper EPM. This EPM contains a well known value. The endpoint mapper activates a free receiving resource and forwards the Connection request to the AP **B** of the device. The Response of the connection contains the AP **B** of the device. After this the AP will be used from that on.

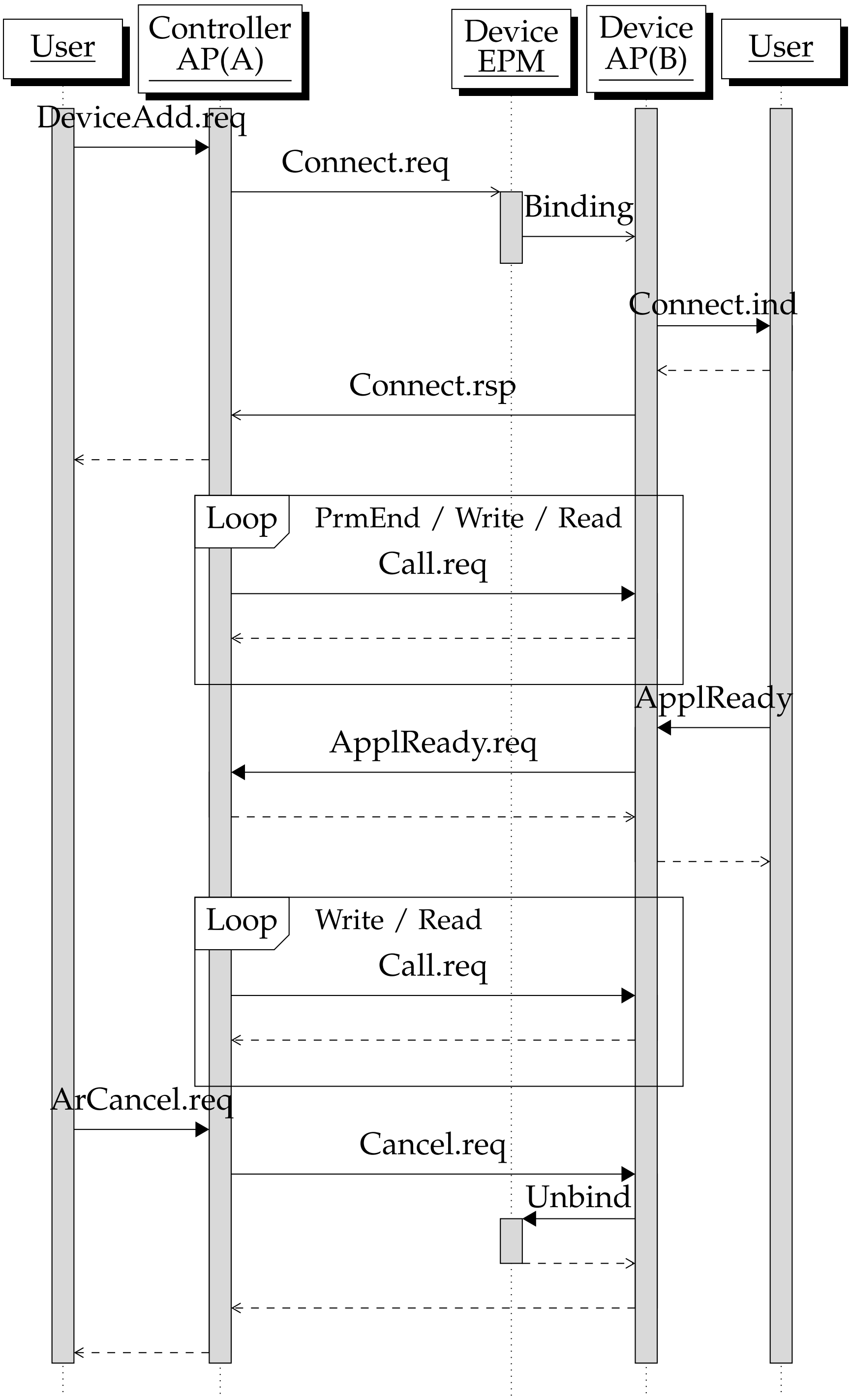


Figure 3.13.: PROFINET RPC Sequence Diagram. The controller interacts with a distributed device over the net. Both have a user that control them and both have an Access Point which identifies them to the net. The Device has an endpoint mapper (EPM) which handles the connect request to the internal Application Relation (AR). The dashed arrows are responses. After connection establishment the controller can read and wirte parameters from the device through the AP. The Device User sends the control request to tell the controller that it is ready. This request can overlap with a call from the controller. With the Cancel request the controller closes the connection and the binding.

After the connection phase the controller initiates requests to the device which are answered and acknowledged with the Call Response.

PROFINET start up

The device is parameterized by the controller through a Write request and tells the device that it has finished with the PrmEnd Request. After the PrmEnd the device needs some time to launch its application. After it is done it sends a
enameControl.req which tells the controller that it has finished. This *Control.req* can be started when the controller has sent a call request. The protocol needs to be able to handle overlap. The cancel request quits the connection. The resources on the device are unbound.

Acknowledgment

Figure 3.13 displays the call requests and responses. A response is also an acknowledgment of a request on a higher hierarchy. The acknowledgment of packets itself is one step below. The network protocols **IPv4** and **IPv6** do not have any acknowledgment mechanism for packages. Higher protocols that are based on IP have to implement it if necessary.

RTA transports payload via the DATA packet type and acknowledges these packets via ACK packet types. A DATA packet can not be an implicit acknowledgment of another one sent before. The ACK packet does not contain any payload. Its only functionality is the acknowledgment of transported DATA packets. **SCTP** provide acknowledgments through the Selective Acknowledgment chunk which acknowledges not only the complete reception of fragments. It supports acknowledgment of single fragments even if there are gaps between transmissions. An example SCTP sequence is displayed in Figure 3.14.

TCP does not differentiate between acknowledgment and payload transportation. Both happen simultaneously. By setting the ACK field it indicates that the acknowledgment number is significant. Every packet may contain x bytes of payload and acknowledge previously received bytes from the communication partner. It is perfect for simultaneous bidirectional transportation. An example of a TCP sequence is displayed in Figure 3.16.

The **IKEv2** protocol does not have transport acknowledgments of the fragments. But every message is acknowledged by a response. To achieve this the response has the same ***Message ID*** as the request. IKEv2 and SCTP both provide a four step initialization phase which includes a cookie mechanism, which is displayed in Figure 3.15.

Excursion: Cookie Mechanism

IKEv2 and **SCTP** provide a **cookie mechanism** which is displayed in Figure 3.14 and 3.15. This cookie transmission guarantees that all requests and responses belong to a single connection where the cookie is defined by the server and function as a nonce. This 32 bit number is unique and guarantees freshness.
This cookie mechanism may prevent retransmission of a packet which belongs to another (already ended) connection but has the same sequence number that is expected for the current call. This might cause an attacker of the net who sends an old packet again to compromise the system or a packet to stay in the net and to be delivered lately by accident at a moment where it is valid.

The **RPC** protocol provides a fragment acknowledgment packet called FACK. No FACK is sent for the last packet. In case of an idempotent request a response is sent and an implicit acknowledge of the sent fragments. In case of an At-Most-Once call the initiator sends an ACK packet again three seconds after it has received the response. This ACK packet acknowledges whole calls and not single fragments.

RPC also provides a **PING WORKING mechanism**, displayed in Figure 3.17. When the responder receives the request and is not able to perform a direct response, the initiator sends a PING. This PING ensures that the responder is still reachable. The responder sends back a WORKING frame. This process gets repeated until the response is computed and sent back.

How do the protocols behave when a failure occurs. It is described in the next section.

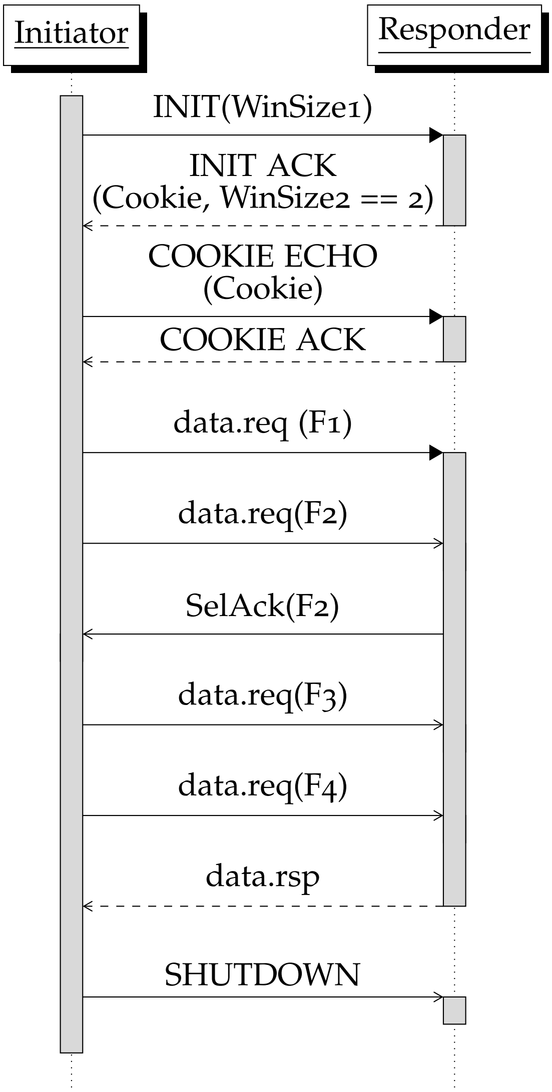


Figure 3.14.: Sequence diagram of **SCTP** for four fragments. Connection gets established through 4 way handshake which guarantees freshness through a cookie. The INIT and INIT ACK packets contain the WindowSize of initiator and responder. The Selective Ack Packet gets send after the second packet.

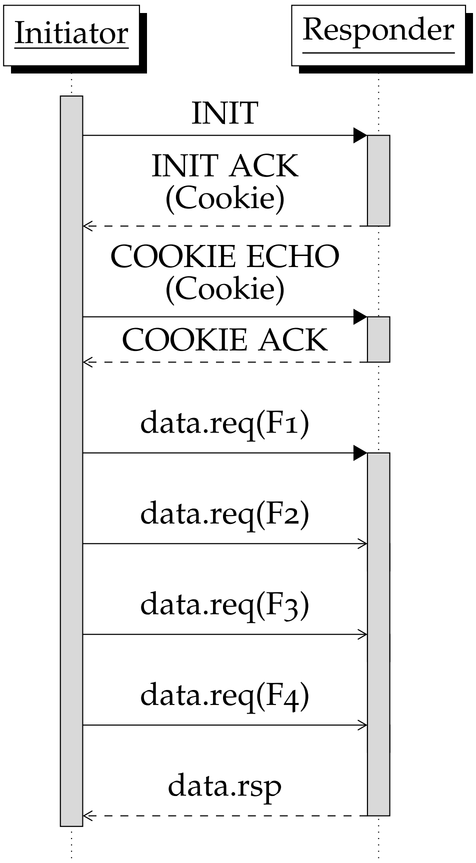


Figure 3.15.: Sequence diagram of **IKEv2** for four fragments. Connection is established through four way handshake which guarantees freshness through a cookie. There is no acknowledgment mechanism for fragments and no cancel mechanism for the connection.

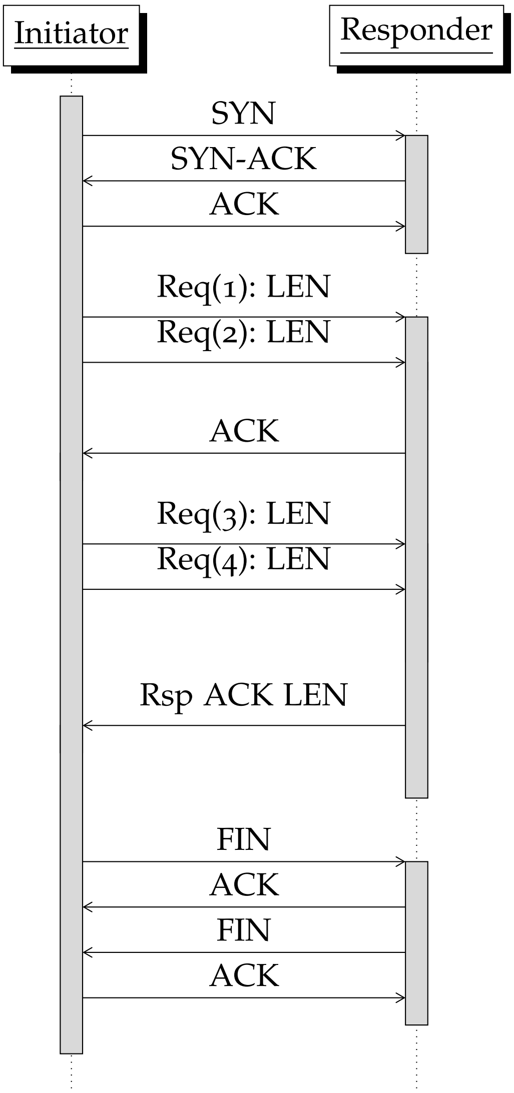


Figure 3.16.: TCP sequence diagram for four fragments. A connection starts with the three-way handshake. The Req(*) and Rsp: is for helping to identify the fragments. It is not coded by the protocol. The ACK says that the ACK number is valid. The FIN flag finishes the connection and can be initialized by both.

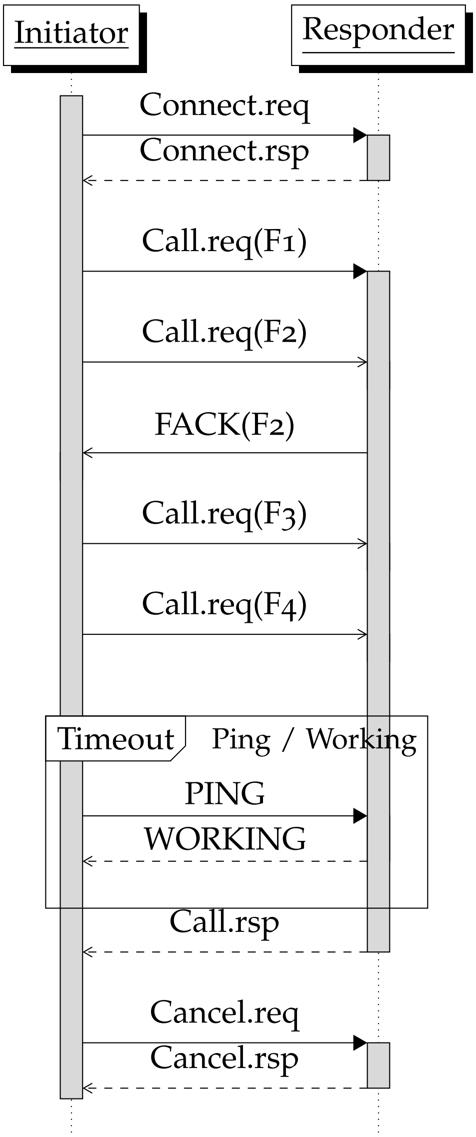


Figure 3.17.: RPC fragment sequence diagram for four fragments. Connection establishment with one fragment call. The call request is acknowledged with a FACK. When the request is running and no response arrives the initiator PINGS the responder who answers with WORKING. This can be looped.

Failure

There are different types of failure: drops, duplicates, delays and crosses of packets.

The **frame drops** need a retransmission for sure because the payload information could not be received. IPv4 and IPv6 do not provide acknowledgment at all and IKEv2 only supports acknowledgment of a call. A frame drop indicates a complete retransmission of the whole call. In TCP only the send window has to be repeated. SCTP and RPC support selective acknowledgment which allows the sender only to repeat the missing fragments.

In case of **frame duplication** the frame needs to be detected as such. A frame duplication may lead to a duplication in the net or by retransmission of the sender even though the original fragment has been received correctly. A receiver may ignore the retransmission or react and tell the sender the current receiving state. TCP and IKEv2 ignore duplications. In case of repeated fragment numbers IPv4 and IPv6 take the content of the newer packets and override the one already stored in the receiving buffer. In case of retransmission this has no impact at all. In SCTP the receiver tells the sender about duplicates in the selective acknowledgment chunk, see Figure 3.8. RPCs fragment acknowledgment do not provide this information.

Frame delays may result in timeouts if they are too long. Every protocol in this comparison use timers but the handling is different. The timer should be as small as possible but big enough that the typical delays do not result in a timeout. TCP has a retransmission timer that sends the whole window again after the specified time. (RFC 793, 1980, p. 10) This leads to a reception of duplicate frames. IPv4 and IPv6 have timers where the complete call has to be received. SCTP use the same timer mechanism as TCP. (RFC 4960, 2007, p. 83) IKEv2 uses a timer for a pending request. If no response is received the request is sent again. The responder does not execute the request again but has to retransmit the response again. (RFC 7296, 2014, p. 25) In RPC there is a retransmission timer for fragments as well as whole requests. The retransmission of fragments is similar to TCP and SCTP. If a duplicate request is detected the handling depends on the class of the request: idempotent, at-most-once or nothing. See Section 3.7 for the description.

Frame overlaps may happen if timeouts occur and a retransmission of a request is started and simultaneously the response was sent before the retransmission is received by the responder. Another possibility is that the responder itself starts a request. This is a scenario for the Application Ready Callback by the responder in CLRPC, the PROFINET component for RPC. If both communication partners start calls simultaneously each retransmission may cause another retransmission. An example scenario is displayed in Figure 3.19. In this example every fragment demands for an acknowledgment which leads to retransmssion of the fragment before. The data is transported correctly but the retransmissions affects performance and net usage. Both sides have to demand an acknowledgment at the same time and the timers have the same value. If there is asymmetry in these variables the normal mode returns.

SCTP may use the selective acknowledgment frame to tell about duplicated transported fragments, described in Figure 3.20. The receiver of the SACK knows that this is a retransmission due to duplicates and does not make a retransmission.

Not only failures may cause an abort also a wrong dimensioning of the resources.

3.8.3. Flow Control

The protocols TCP, SCTP and RPC support acknowledgment of fragments and flow control as well. Flow control describes the ability of a receiver to tell the sender how many fragments it is able to receive until they have to be acknowledged. This amount of fragments is called the window size.

TCP sends the window size in every packet where the ACK value is significant. (RFC 793, 1980, p. 4) So through the 3-way handshake of TCP it is guaranteed that the window size is known to Initiator and responder.

SCTP has an **Advertised Receiver Window Credit** that is transported in the INIT chunk by the initiator. The responder tells the sender its window size through the INIT ACK chunk. This value may be changed during an established connection with SACK chunks. The included window size field

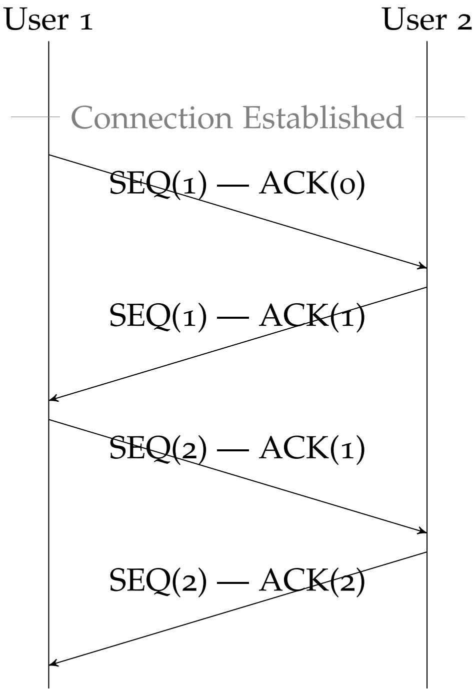


Figure 3.18.: TCP without intersection. Connection established, payload length is always 1, ACK is always valid. No intersection.

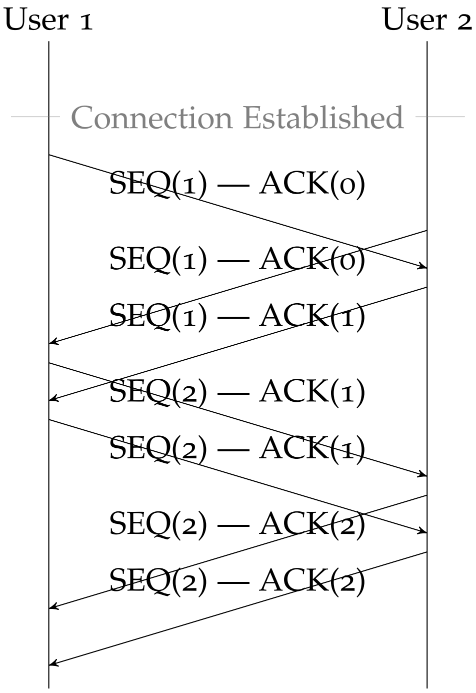


Figure 3.19.: TCP with intersection. Connection established, payload length is always 1, ACK is always valid. First intersection occurs followed by a retransmission of the first fragment of user 2. This retransmission occurs, followed by another retransmission of the second packet from user 1.

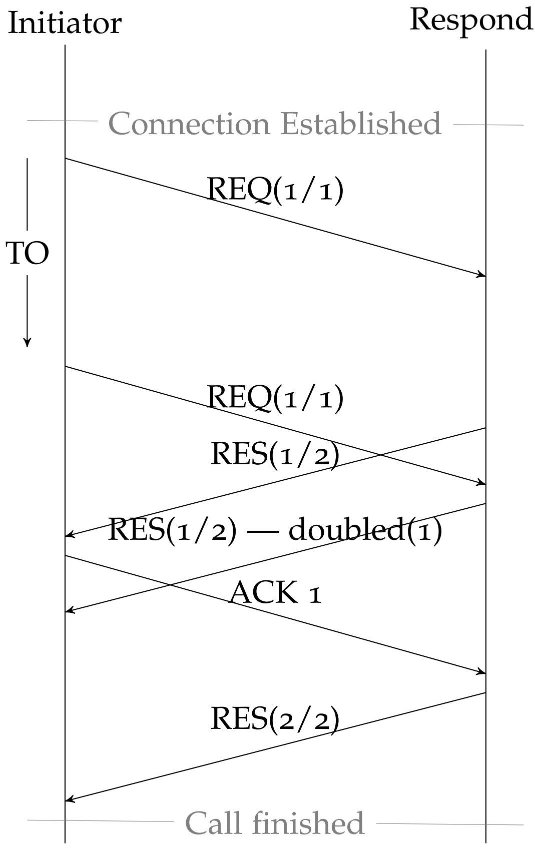


Figure 3.20.: SCTP call with intersection. ACK after every fragment. Single fragment request has been received. Computation of response takes time. Timeout occurs and retransmission is received after sending the first fragment of response. The responder sends the first response fragment again but adds a selective ACK field with information about duplicated fragments. User 1 detects retransmission and sends only one ACK.

specifies the value of receiving buffer space at the moment the SACK is computed. With the additional information of acknowledged fragments and gaps in it, the receiver of the SACK knows which fragments are needed and possible to transport correctly. (RFC 4960, 2007, p. 81)

SCTP and TCP transmit the window size in bytes. (RFC 4960, 2007, p. 8) (RFC 793, 1980, p. 4) **RPC** tells the window size in fragments. (CAE Specification, 1997, p. 585) To use bytes over fragments makes sense if the payload length varies. The cause may be an unknown MTU that varies or in case of SCTP the chunk architecture of the protocol. If more than one chunk is transported in a packet the length of a chunk type's payload varies. The simple knowledge of the amount of chunks that are still coming is not useful and often not possible to compute. The sender does not know how many other chunks will get transported simultaneously.

Linked to the flow control feature is the used memory of a protocol.

3.8.4. Memory

This chapter describes the amount of memory the protocols need for the fragmentation process to store and process the state. This comparison makes the assumption that the packets have sequence numbers with the size of 32 bits, a maximum window size of eight, the length of data has a maximum of 65546 bytes and the maximum payload size is 1400. The used memory for timers is not described in any of those protocols. So the 24 bytes of the RTA timer are used as reference. The comparison is displayed in Figure 3.21. It does not contain the memory for the data buffer and information about duplicate fragments.

In **TCP** the whole protocol is designed to transport a bytestream in fragments from one node to another. (RFC 793, 1980, pp. 19,21) Every sequence number of the sending and receiving partis described per byte. It is bidirectional and there is no variation between initiator and responder.

IPv4 differentiates between sender and responder. The sender works sequentially and sends each packet after the other. Since there is no acknowledgment mechanism the responder needs to remember which bytes it has

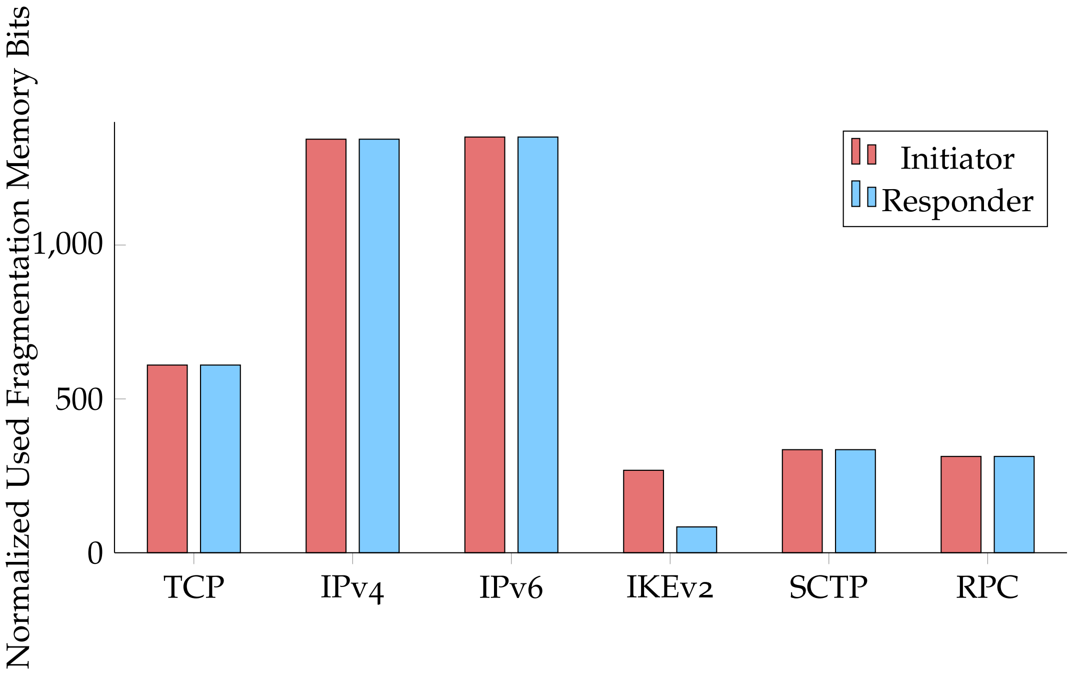


Figure 3.21.: Comparison of memory used for fragmentation for the initiator and responder of a call. Memory is adapted for sequence numbers of 32bits, a max. window size of eight, 65546 bytes of max. data length and max. payload length of 1400 bytes. IPv4 and IPv6 need fragment bit tables for the whole data in eight octets. TCP, SCTP and RPC only save information about the current window. IKEv2 only needs a timer for the sender. Request is repeated if response is dropped. RPC's selective ACK needs less memory than SCTP's. The full description is in Appendix H.

already received and which are are expected to be received. Therefore IPv4 has a *Fragment Block Bit Table* which has a bit for every eight bytes that has been received. For a maximum of 64KBytes of data it has a size of 1kBit. (RFC 791, 1981, 26ff) **IPv6**'s need for memory is quite similar to IPv4. (RFC 2460, 1998, 21ff)

IPv4 and **TCP** have an accurate description of the memory bits. The newer standards **IPv6**, **IKEv2** and **SCTP** do not give information and describe it instead in prose. For theses protcols there are assumptions about the technical fragmentation realizations of the data that is partially based on the other standards and about saving the information on an optimum way.

For **IKEv2** used memory is not directly described. The values in this comparison are optimistic and stand for the lower bound that is possible. In IKEv2 there is on offset field and so the fragments have a fixed packet size. On the receiving part there has to be a received fragment table. With the reception of one fragment, if it is not the last one, the size of each fragment is computable. Instead of 1024 bits for the received bit table the fragment table has the size of 46 bits.

SCTP supports acknowledgment and needs the same ressources for request and response. The memory for different streams is not included. Both nodes need timers, information about received fragments and gaps. The fragmentation mechanism of **RPC** is similar with small differences to it. The selective acknowledgment of RPC needs 27 bits less space than the one in SCTP. (RFC 4960, 2007, 34ff) (CAE Specification, 1997, p. 585)

IPv4 and IPv6 only need a timer for the receiving part. SCTP and RPC only for the sending part. In all these protocols the initiator and responder send and receive (request and response) so all resources are needed on both sides. Only in IKEv2 the sender has a timer that triggers a retransmisson.

The comparison and analysis of existing fragmentation strategies is the basis for the next step: Defining the RTA v2 protocol with included fragmentation.

4. Protocol Design

Based on the comparison of fragmentation strategies the RTA version 2 has to be designed. Version 2 should support fragmentation. RTA v1 is defined in an IEC norm (IEC 61158-6-10, 2015, pp. 231 sqq.). This chapter will outline the changes that are necessary to describe the new v2. Three Sequence Variants are described: The Basic Variant where records and alarms are transported completely in parallel, the Advanced Variant for a common window of alarms and records and the Chunk Variant for transportation of alarms and records in one packet.

The packet structure, the sequence diagrams and the underlying state machines are described and explained below.

4.1. Packet Structure

The packet structure of RTA v1 is displayed in Figure 3.11. The draft of the new protocol is displayed in Figure 4.1. The existing elements of v1 shall not change their meaning in v2 but they are extended and renamed.

The ***Destination AlarmRef*** and the ***Source AlarmRef*** of v1 are renamed to ***Destination SAP*** (Service Access Point) and ***Source SAP*** since v2 is not limited to transport alarms. Service Access Points have the same functionality by defining endpoints but they are renamed to a more general term.

The ***Protocol Version*** is set to 2. The Protocol Description Unit ***PDU Type*** describes the kind of payload. The new types for v2 are described in Section 4.1.1.

The ***Flags*** may be expanded. The need of additional flags is discussed in Section 4.1.2.

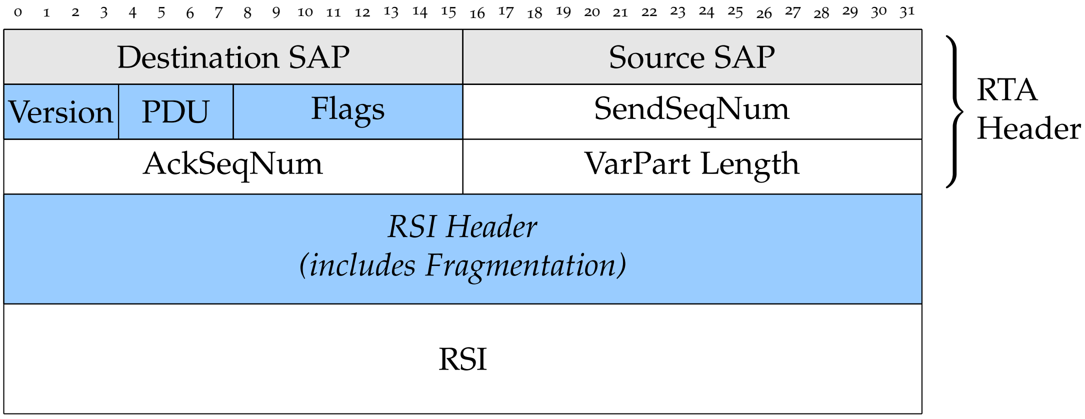


Figure 4.1.: Marked changes of RTA header version 2 in contrast to version 1. The ***SAP***s are renamed. The ***Version Type*** and ***Flags*** are extended and the ***RSI Header*** contains the fragment information.
⬜ : Renamed
🟦 : Changed

The ***SendSeqNum*** and the ***AckSeqNum*** will be used in a similar way as before. The initialization values stay the same: $0xFFFF$ for ***SendSeqNum*** and $0xFFFE$ for ***AckSeqNum***. With every new fragment to send the ***SendSeqNum*** is incremented. The ***AckSeqNum*** contains the information of the highest received ***SendSeqNum*** of the communication partner. The sequence numbers contain no mapping to a specific call.

The ***VarPart Length*** describes the payload length of RTA which begins right after this field. The allowed range of v1 from 0-1432 bytes endures but it is narrowed when an RSI header (with fragment information) is necessary.

The ***RSI Header*** contains the information of the Remote Service Interface which will include fragmentation information.

4.1.1. PDU Type

The **Protocol Description Unit (PDU)** Type describes the kind of payload RTA transports. In v1 there are three types: **DATA** for alarms, **ACK** for acknowledgments and **ERROR** to tell the communication partner about the

Table 4.1.: PDU Types compared to protocol versions and different implementation possibilities 2a, 2b and 2c. In v2a **DATA** type is used for alarms and records. V2b use **FRAG** for records and v2c distinguish between request **FREQ** and response **FRES**.
🟦 : Used type in this variant.
░ : Optional usable for alarms.

PDU Type	Value	Version 1	Version 2a	Version 2b	**Version 2c**
DATA	1	🟦	🟦	░	░
NACK	2				
ACK	3	🟦	🟦	🟦	🟦
ERROR	4	🟦	🟦	🟦	🟦
FRAG	5			🟦	
FREQ	5				🟦
FRES	6				🟦
Reserved	7 - 15				

abort of the connection. In Table 4.1 version 1 and version 2 are compared concerning the PDU Types. For version 2 of the protocol there are three different variants a, b and c.

The type **NACK** has been used to tell a sender of an alarm a negative acknowledgment. This type is no longer being used in version 1 anymore.

In Version 2a no new **PDU** type is used. The **DATA** type is used to transport the records for version 2 of the protocol. Alarms have to be sent with version 1 of the protocol. Variant a would permit to send alarms and records in one window. The ***Protocol Version*** field would be the indicator for the next protocol which is not the role of this field.

In Version 2b there is an extra type **FRAG** for the records. Alarms and records are distinguishable by the PDU Type.

Version 2c extends b by the use of an extra **PDU** type for the request which is sent by the initiator and for the response which is sent by the responder. This distinction helps the developers and supervisors who track the packets to match them to sender and receiver. **Version 2c** is used for all sequence realizations.

4.1.2. Flags

The flags of RTA v1 are defined as follows:

0	1	2	3	4	5	6	7
WS	Reserved			TACK	Reserved		

In v1 the **window size WS** is 1 but Bits 1-3 are reserved for increasing the window size in newer versions of the protocol. The RPC protocol in PROFINET uses a fixed window size of 2 to send records. There are two receive resources reserved for application relations plus one for an implicit read access on the device. This kind of read access does not need an established connection. After the response has been received it is disconnected again.

The default window size for RSI will be 2 as well. If the controller or device support more receive resources a window size of up to eight is possible. Four bits are needed. In v2 all three reserved bits are used to increase the window size.

Bit 4 is the **TACK** flag which indicates a demand for an **acknowledgment** of the highest received sequence number without a gap. The sender will send the next window when the ACK PDU has been received or a timeout has occurred and it has to send it again. RPC has the Flag **NO FACK** which is coded the opposite way. It signalizes the sender that no acknowledgment is wanted. It has the same functionality as the **TACK** Flag.

For v2 additional flags are necessary. Four variants of how the RTA flags can be coded are displayed in Figure 4.2.

In Variant 1 the **MF** flag indicates that more fragments are following. During a fragmented transportation this flag is always set except for the last fragment.

Variant 2 indicates that the fragment is the first one, **FF**, and/or the last one, **LF**. This provides extra information if it is a fragmented call or not. If both are set to 1 it is a single fragment.

	0	1	2	3	4	5	6	7
V.1 {	Window Size				TACK	MF	Reserved	
V.2 {	Window Size				TACK	FF	LF	Reserved
V.3 {	Window Size				TACK	Reserved		
V.4 {	Window Size				TACK	DF	Reserved	

Figure 4.2.: Flags implementation possibilities. The windows size is coded in 4 bits. The TACK flag . ***Variant 1*** has a More Fragments **MF** Flag which indicates it is not the last fragment. ***Variant 2*** has a First Fragment **FF** and a Last Fragment **LF** flag that indicates the first, middle (no flags set) or single fragment (both set). ***Variant 3*** has no flags, control information in fix fragment header. ***Variant 4*** has a Do Fragmentation **DF** flag that indicates an optional fragment header is added between RTA header and RSI Block.
🟦 : Changed

In Variant 3 there is no fragmentation control flag at all. This is possible if the information about the transported fragments and the information if a fragment is the last one is coded in the fragment header.

Variant 4 contains the Do Fragmentation **DF** flag which indicates that there is an optional fragmentation header which is only there if the call really has to be fragmented. The signalization of an optional header is unusual. IPv6 and SCTP signalize an optional header with an 8 bit ***Next Header*** value which codes multiple headers. For that realization the fragment information has to be separated from the fixed part of the RSI header.

Variant 1 is used for the Basic- and Advanced Variants. It is the simplest way to indicate fragmentation for a fixed fragment information coding and there is no extra place necessary for new flags in another place in the header. Variant 3 is used for the Chunk Variant because every control information is stored in the chunk it self and there is reserved space in every chunk header for protocol specific information.

4.1.3. Fragmentation Information

The fragmentation information can be coded in an extra fragmentation header, in the fixed part of the RSI header or hard coded in the RTA header.

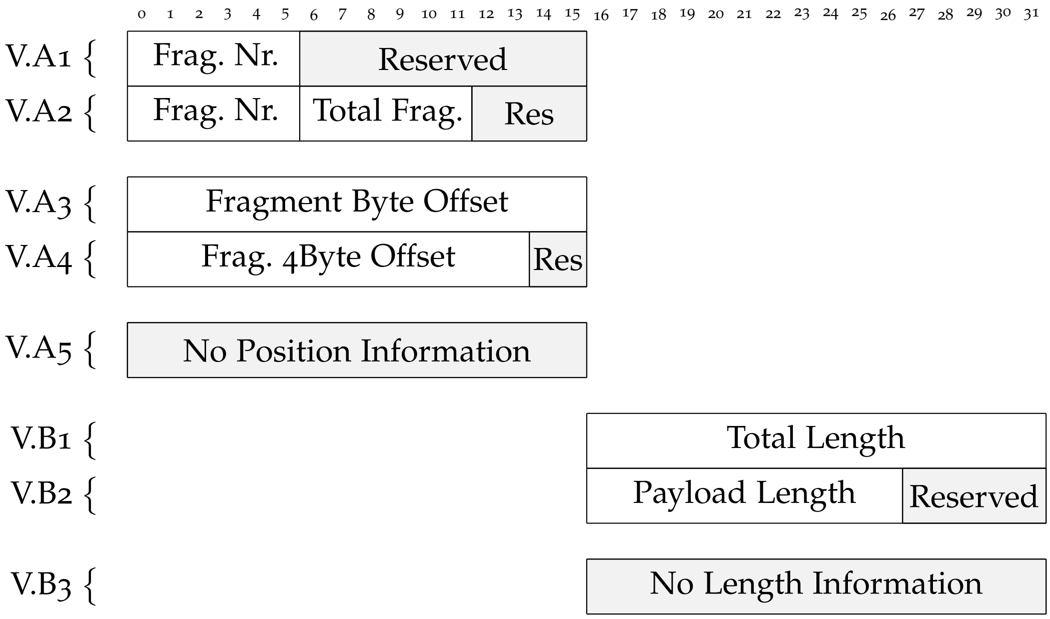


Figure 4.3.: Fragmentation Header realization possibilities. First 16 bits for position information. Next 16 Bits for length information. Variant A1 identifies the fragments with numbers beginning from 1. A2 has additional information about total number of fragments. A3 contains the position in a byte offset and A4 a multiple of 4 bytes. A5 has no position information. The Total Length B1, the Payload Length B2 or no length information may be transported.

The fragmentation information covers the offset position of the fragment and the length information of the **Data** and/or **Payload** of the fragment. Different variants for coding the fragmentation information are displayed in Figure 4.3.

The **Offset Information** can be coded as a ***Fragment Number*** which starts with one and ends with the last fragment. For the maximum ***Data Length*** of 64 kBytes the maximum ***Fragment Number*** is 46. The ***Total Fragments*** field tells the sender how many fragments will be sent. This information helps the receiver to abort the transportation at the beginning of the call if its receive buffer is too small.

In Variant A3 the offset is coded as byte offset. Variant A4 decodes it as a four byte offset and permits a ***Payload Length*** which is not dividable by 32. The maximum ***Payload Length*** is 1400 Byte.

The offset coding has the benefit over the fragment number that the payload order can be mixed up by the sender. If the receiver supports random reception of fragments it has to use a buffer byte table where the bytes which have been received are marked. The pointer to the information where the payload data is copied to the local receiver buffer is double checked. The receiver knows which part of the data it expects next and compares it with the offset information in the header, which is another positive affect.

There is no need for position information at all Variant A5. If only in order transmission is allowed the sequence numbers and the ***More Fragment*** bit are enough to determine the offset in the receiving buffer to insert the new payload.

In Variant B1 the length of the payload, in B2 the ***Total Length*** of the ***Data*** and in B3 no length information is coded. Both length are not necessary. The ***Payload Length*** is not necessary to transmit. It can be computed through the ***VarPartLength*** field minus the length of the fragment header when no chunks are used. If the fragmented part is transported in chunks the ***Payload Length*** is necessary. The information about the ***Total Length*** of the Data is not necessary. The receiver may abort when the receiving buffer would be exceeded. Without chunks the ***Payload Length*** is redundant information.

For the Basic Variant of RTA v2 the offset is coded in bytes. It supports a check on the receiving part if the pointer to the part in the buffer is correct and if a shorter field have been used bits would be reserved. For the Chunk Variant the length information of the chunk is coded in the chunk header.

4.1.4. Implementations

There are three different implementation possibilities. The addition of fragment information as a permanent layer between RTA and RSI (1, Figure 4.4), an optional fragmentation header (2) or an optional fragmentation chunk header (3, Figure 4.6).

In Variant 1 the fragment and RSI information are fixed parts of the RTA header. The flags includes the ***More Fragment*** flag. The fragment informa-

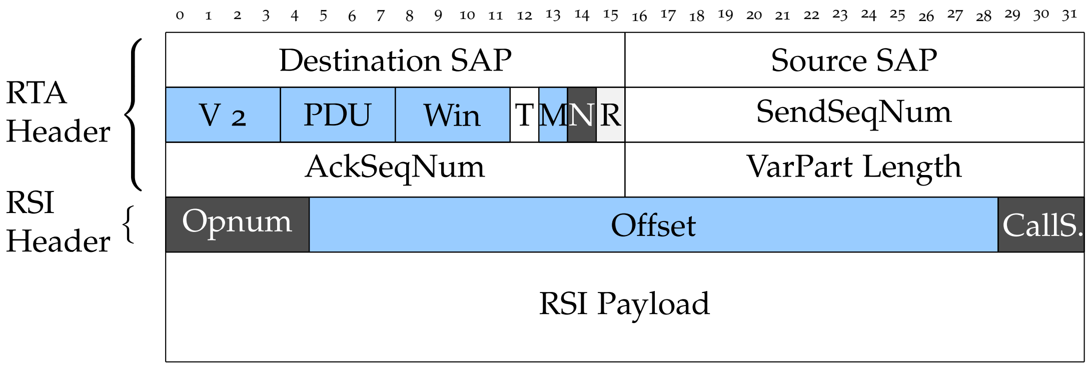


Figure 4.4.: Packet Variant 1. RTA with fix RSI Extension. The flags contain the additional More Fragment Flag **M** and the Notification Flag **N** which tells the initiator that the responder is ready to send a Notification to the initiator. The RSI header contains the Opnum, the byte offset and the Call Sequence. The payload is a fragment of the RSI call.
🟦 : Contains fragmentation information
⬛ : RSI Information

tion may theoretically be transported with every version that is displayed in Figure 4.3. The version without an extra fragment number or fragment offset has been used since the sequnce number is enough to identify the fragment of the RSI payload. The protocol is designed to always use the maximum possible ***VarPartLength*** which is 1432 . The maximum ***Payload Length*** is 1400. Therefore the buffer offset for the receiving part is always a multiple of 1400. Variant 1 is the easiest and simplest way of including fragmentation functionality.

In Variant 2 there is additional fragmentation information in the RSI header. A PDU type FRAG (PDU type v5) or **FREQ** and **FRES** (PDU type v6) signalizes that all fragmentation information is in the fragment header. The first element indicates which optional header or which payload comes next. This makes it easy to add other additional headers later.

In Variant 3 the architecture of the protocol is based on chunks where more chunks can be transported in one RTA frame. The Flags do not contain any extra flags for fragmentation. The PDU type indicates with the types **FRAG** or **FREQ** and **FRES** that this is the next chunk which follows. With the type of DATA it indicates that an Alarm will follow. Alarm High has its own

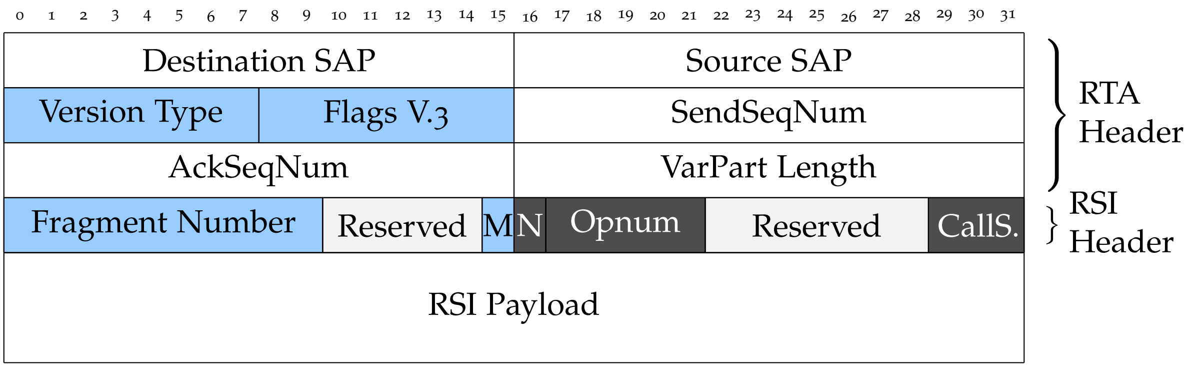


Figure 4.5.: Packet Variant 2. Optional fragmentation header realization. In addition to the first version the fragment number is included and the More Fragment bit is included in the RSI header. The size of the RSI header is the same.
🟦 : Contains fragmentation information
⬛ : RSI Information

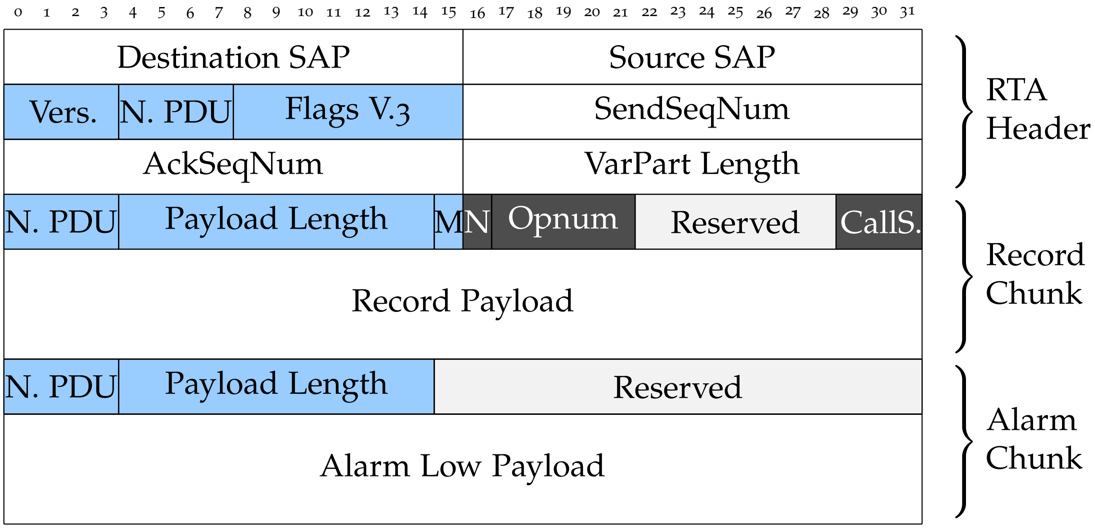


Figure 4.6.: Packet Variant 3: Chunks. RTA header is extended with a chunk type field that indicates that a fragment chunk follows. The chunk type indicates the following chunk that comes next. The Payload Length is considered as part of this chunk. In the RSI chunk header the M Flag is the More Fragment Flag and the N Flag for the Notification bit of the responder. The Opnum is the operation number of RSI and the CallSequence is used for every new call. The last chunk type is 0 and indicates that nothing follows.
🟦 : Contains fragmentation information
⬛ : RSI Information

PDU type. Alarms and records use the same receive resource management with a common window size. For the **chunk type** in the chunk headers the ***PDU type*** field is used. The payload length is the length of the payload data of each chunk. This is necessary to know in order to get a pointer to the next chunk header.

How will the protocol be look like on the wire? The next section describes the sequence.

4.2. Sequence

The versions that are described in Section 4.1.4 lead to three different communication sequences. In the basic variant the alarm and RSI parts are separated into different state machines. The sequence and acknowledgment numbers are not linked, the window size is separated but the alarm reference of an alarm addresses the same endpoint as the SAP of RSI does. An extra resource for Alarm High and Alarm Low has to be reserved besides the resources for RSI. The communications are not influenced by the other.

Alarm Priorities

There are two alarm priorities: **Alarm High** and **Alarm Low**. The high one receives preferential treatment and is used for critical events. The low one is used to reset a connection after timeout or user trigger. The different priorities are identifiable through a different Frame ID in the PROFINET Ethernet header.

There could also be a possible Advanced Variant which combines the state machines of alarms and RSI in the same transport flow. This includes the sequence and acknowledgment numbers and a common window size. Records and alarms would be transported with a common ***Frame ID*** and with the protocol version 2. These advanced variants differ if they use chunks or not.

RSI uses two different timers for records. The fragmentation timer **TO** and the call timeout **TOC**. The fragmentation timer ensures that fragments are acknowledged and the call timeout ensures that the initiator does not have to wait for the response for an infinite time. The **TO** timer starts with the first fragment of the request and ends with the receiption and correct defragmentation of the response. The alarms have a separate timer **TOA** which has to ensure that the alarms are transported. The alarm timout **TOA** is 100ms, the fragmentation timeout **TO** is 2s and the call timeout **TOC** is 3min.

4.2.1. Basic Variant

The Basic variant does not include alarms in the same transportation window. Alarms are sent in parallel and have their own receive resources. An example transportation is displayed in Figure 4.7. The sequence diagram is separated in different sections in order to show the cases that may appear. All cases are displayed in Appendix A.

The controller is always the initiator of a call and starts a request. The device is the responder. The first fragment always uses the window size 1 which has two reasons. First, the controller does not know the window size of the device and second, the Service Access Point **SAP** is not known. So the first fragment is addressed to the endpoint mapper. The first fragment that is repeated contains the **SAP** for this connection and the window size of the device. The window size shall not be adapted during the connection. When a timeout occurs the initiator repeats the message three times before it aborts.

The meanings of the states in the sequence diagrams are:

- **Start (empty)**: Is the first fragment of a connection.
- **A(x)**: x fragments of the request have been acknowledged.
- **A(x)R(r)**: x fragments of the request have been acknowledged but an fragment timeout has occurred and the number of maximum timeouts has not yet been reached.
- **W/R(r)**: All fragments of the calls have been acknowledged but no response fragment has been received by the initiator. There are r timeouts left until the connection will be aborted.

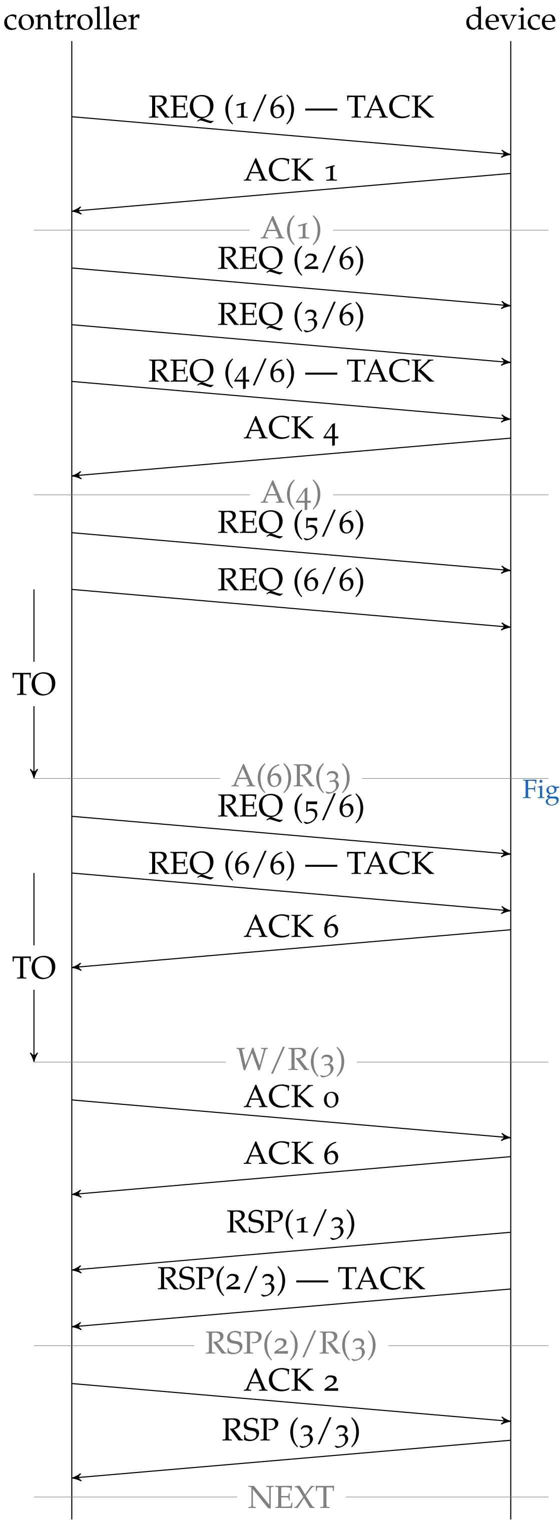


Figure 4.7.: Example sequence diagram 1 of basic variant without frame drops. Server window size for request is 3 and client window size is 2. First fragment is first connect fragment with window size 1.
Last fragment of window contains TACK flag. Last window of request or response does not contain TACK on first transmission. Next transmissions set the TACK until device has acknowledged every fragment. The ACK 0 is a PING signal which demands for a a working signal.
Server starts the response with the controller window of 2. Client acknowledges every response window.
The grayed lines are linked to sequence parts that are used to describe all parts and failure conditions as illustrated in Appendix A.

- **RSP(x)/R(r)**: x fragments of the response have been received and r timeouts are left until the connection will be aborted.
- **NEXT**: All fragments of the response have been received. The next request can be sent.

The frame description is structured like this:

- **REQ(m/M) – TACK**: m^{th} fragment of a request with M fragments ins total. TACK signalizes if the flag is set.
- **RES(n/N) – TACK**: n^{th} fragment of a request with N fragments ins total. TACK signalizes if the flag is set.
- **ACK x**: Acknowledgment of the packet where x is the fragment number of the opposite site's call.

There can be several possibilities of what could go wrong. Frame drops may occur. Two example scenarios of frame drops are displayed in Figure 4.8. These frame drops may lead to retransmissions. If the sender of a window does not receive an acknowledgment the window will be sent again.

Another possibility of what could go wrong is the overlap of packets. An example sequence with a fragmented request and response is displayed in Figure 4.9. The last two request fragments are received correctly but the computation of the response takes time and so a timeout occurs. Right before the response has been received by the controller the last two request fragments are retransmitted. Because of this retransmission also a retransmission of the first two response fragments from the device occurs.

4.2.2. Advanced Variant

The Advanced Variant of RTA v2 uses shared resources for alarms and for records. The sequence numbers, the acknowledgment mechanism and the window size take effect for both kinds of data. The Advanced Variant extends the Basic Variant with the possibility to transport alarms while transmitting a record. Therefore, the sequence diagrams are extended by extra sections.

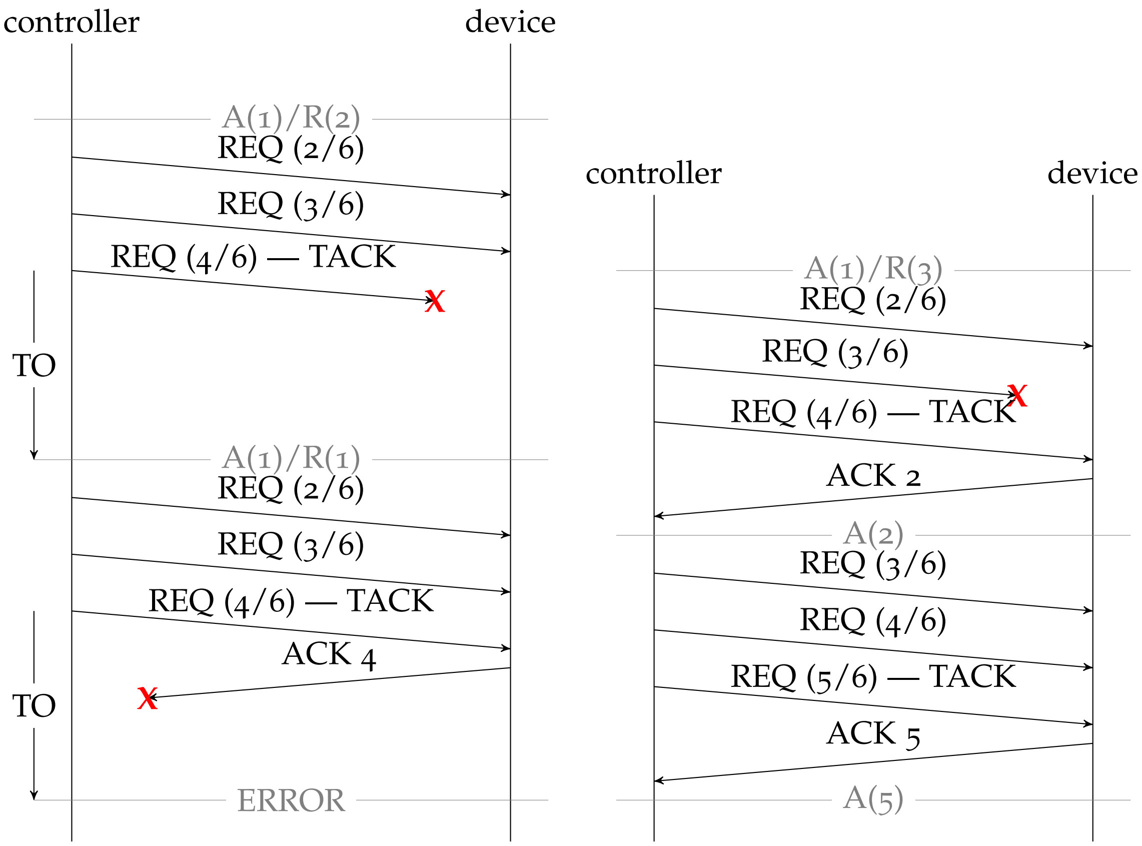


Figure 4.8.: Framedrop requests. If requests with TACK or ACK are dropped a timeout occurs and a retransmission of the window starts. After three retransmissions an error occurs and the connection is terminated. If a frame is dropped during a window the responder acknowledges the last fragment received in order.

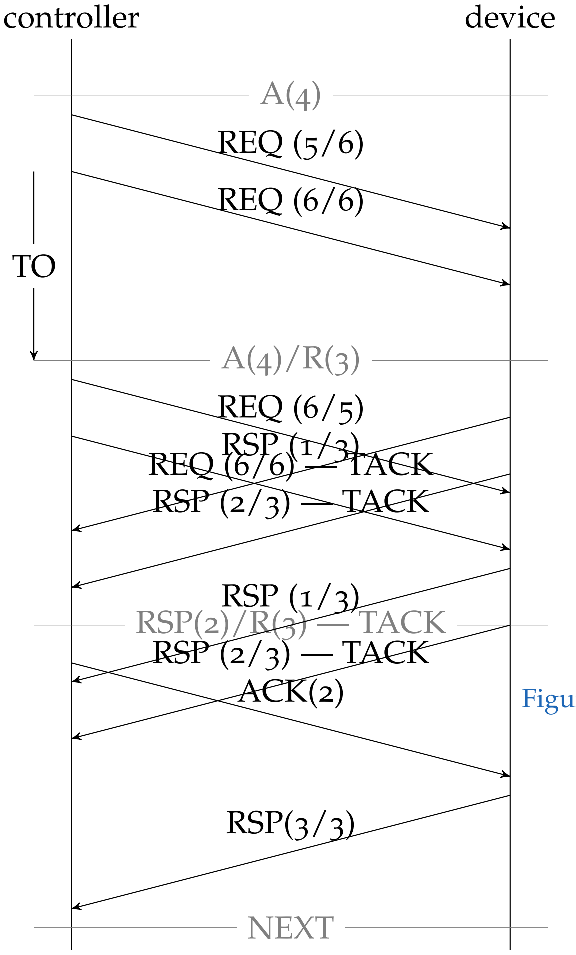


Figure 4.9.: Example sequence diagram of basic variant with intersections. Window size for controller and sever is 2. Response and retransmission of request overlaps which causes retransmission of response fragments. The TACK flag is ignored for response retransmissions.

The alarms use the **TOA** timeout, which is 100ms. In case of a running call the initiator decreases the timer from the fragment timer value **TO** to the alarm time value **TOA**.

The meanings of the section states are extended by additional information about sequence numbers:

- **Seq(y_y)...**: The section that follows ... is extended with SendSeqNum of controller (x) and device (y).
- ***Seq(x_y)**: Is valid for all basic sections and adds the information of SendSeqNum of controller (x) and device (y).

In Figure 4.10 an example sequence for the transportation of alarms is displayed. The RSI initiator and responder is allowed to send an alarm with the priority high and low at all times. Every alarm has to be acknowledged immediately. The TACK field is set in every alarm PDU. When an alarm has been sent a timer is started. After the Timeout Alarm **TOA** the alarm is resent. If there is no acknowledgment after three retransmissions of the alarm the alarm sender aborts the connection.

The case where the controller has to send alarms to device during the fragmentation process of a request is displayed in Figure 4.11. The window sizes are two for both nodes. The left sequence displays the case without frame drops and the right sequence displays the same case with frame drops. The behavior of the protocol would be the same if the initiator fragments the response and the device has alarms to send.

When the Alarm Low **AL** event occurs there is still one frame left in the window where the Alarm Low can be sent directly. The next alarm which occurs **AH** can only be sent directly when the last window has been acknowledged. In the right sequence diagram the framedrop of the acknowledgment causes a retransmission of the last window. The Alarm High event on controller side starts a timer.

The case where the device has sent all response fragments but the last window has not been acknowledged and an alarm occurs is displayed in Figure 4.12. There is no acknowledgment information for the device if the last response fragment has been received correctly or if some frame has been lost. The controller has a Fragmentation Timer **TO** that sends an ACK if not

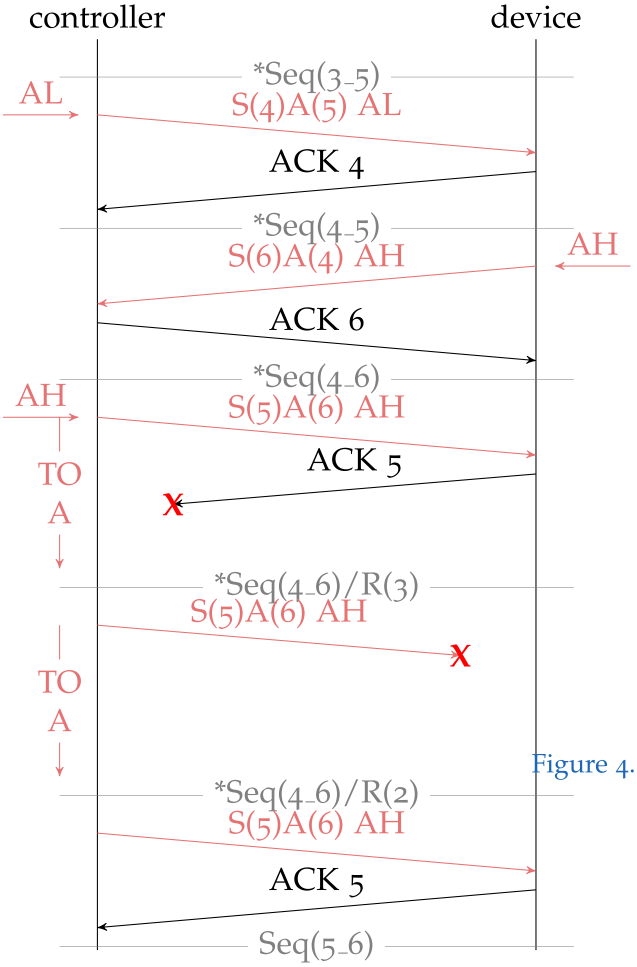


Figure 4.10.: Example sequence diagram of Advanced Variant with alarms only. No overlap of an RSI call and no overlap without controller and device are allowed to send Alarm Low **AL** and Alarm High **AH**.

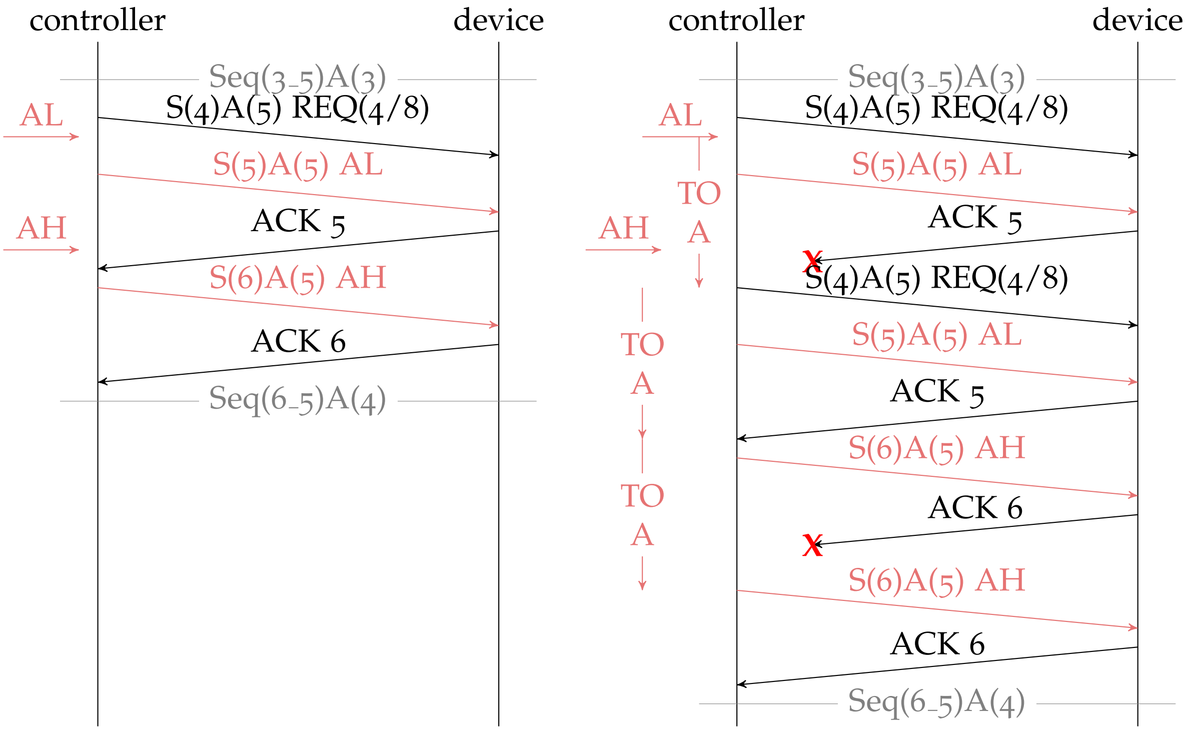


Figure 4.11.: Example sequence diagram of Advanced Variant with alarms and records. Alarm Low **AL** fits in window of size 2 and can be sent instantly. After timeout **TOA** window is retransmitted. Alarm High is queued and is sent after the previous window has been acknowledged.

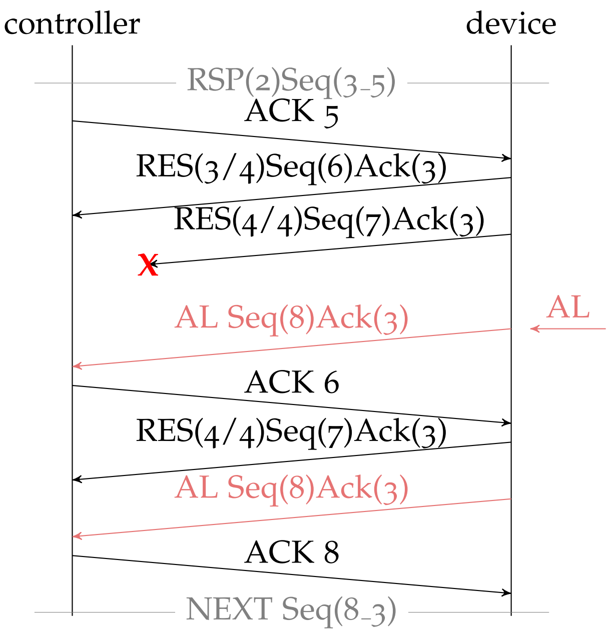


Figure 4.12.: The server sends an alarm after he has finished his response. The alarm is sent but can not be delivered. The device starts a timeout and retransmits the last window. After acknowledgment of the RSI window the alarm is transported.

all expected fragments could be received. If an alarm occurs on the device side the alarm is sent and the Alarm Timer **TOA** is started. The alarm is not accepted by the controller. The second last fragment is acknowledged so that the device can send the last response fragment and the alarm in one window.

In case of an alarm that has to be transported by the currently receiving part of a record connection it can be sent immediately. A scenario with two alarms is displayed in Figure 4.13. The transport process of RSI is not influenced. If the alarm has been received an ACK is sent. The Alarm High has been received by the controller during the transportation of a window. The next fragment REQ(7/8) acknowledges the **AH**.

If the first alarm is lost then it is resent after the alarm timeout is triggered. Another alarm AH is sent in the previous window. After the timeout both alarms are resent simultaneously. A separate timeout of alarm low and alarm high would trigger the retransmissions of both, alarms separately and guarantees that the alarm high is not retransmitted immediately after the initial send. This reduces the sent frames if the acknowledgment of the alarm low is lost. If the alarm is lost separate timers do not bring any benefit. They may even delay the transport of the second alarm since the second alarm can only be processed if the first has arrived.

The acknowledgment packets (ACK) have been left out of the window calculation. An argument to include them would be that ACK packets use the resources as well and so have the ability to block them. A counter argument is that the blocking time is much shorter because no payload is included and the information can be extracted faster.

4.2.3. Advanced Variant with Chunks

The Advanced Variant can be extended by the use of chunks. Chunks are used to transport more than one different payload type in one frame. The packet structure of this variant is described in Figure 4.6. It is able to transport an alarm HIGH, LOW and a record fragment in one packet. This technique reduces the potential of the alarms thwarting the transportation of records.

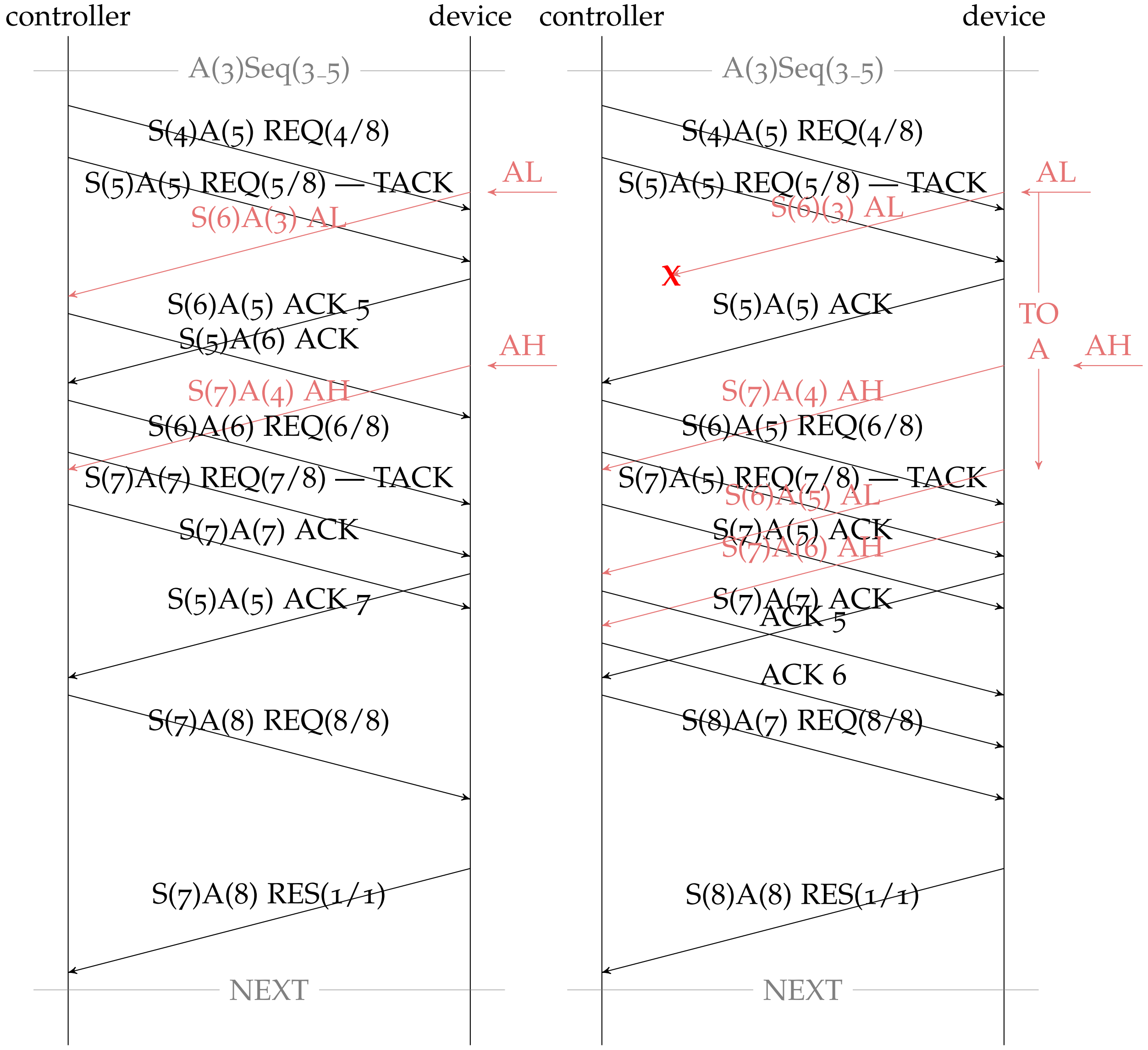


Figure 4.13.: Example Sequence Diagram of Advanced Variant with alarm and record from different nodes. The left sequence illustrates the sequence without and the right with a frame drop. The transport of the request is not deferred. On retransmission both alarms are transported in one window.

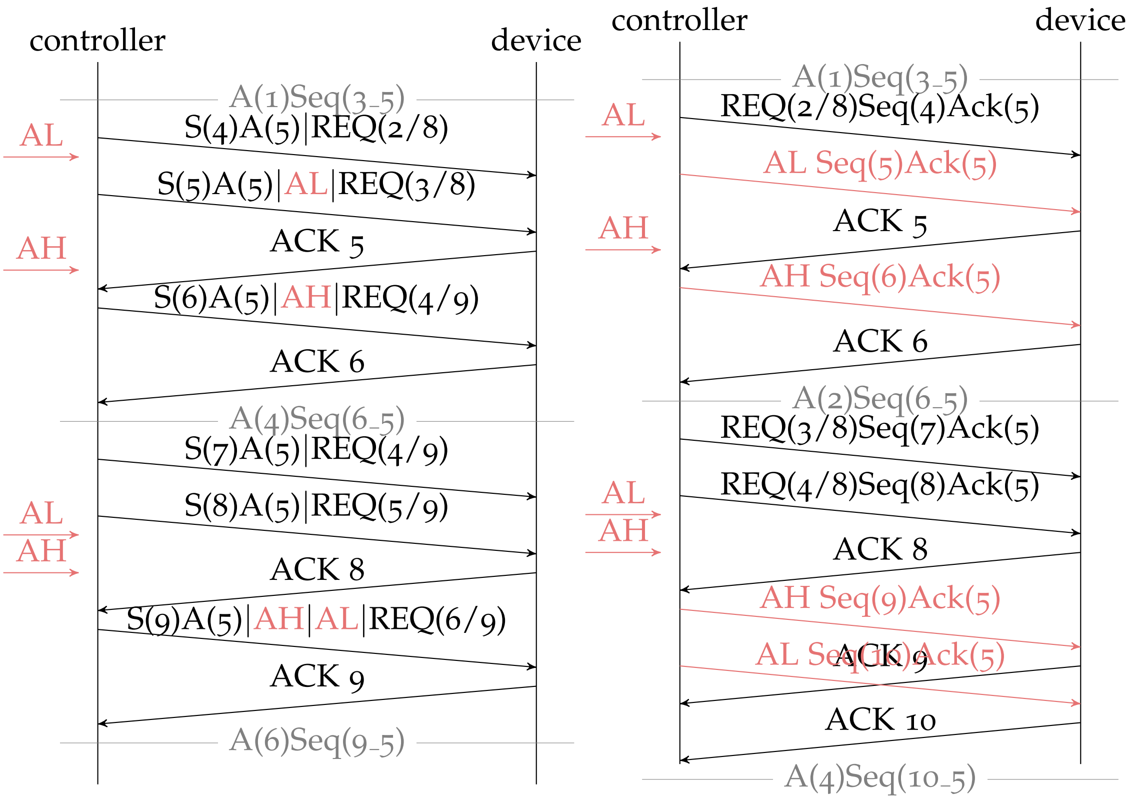


Figure 4.14.: Example sequence diagram of Advanced Variant with chunk realization on the left and without chunks for the same scenario on the right. Chunks are separated with |. S(*)A(*) signalizes sequence and acknowledgment number. Total number of RSI fragments is increased from 8 to 9.

This benefit takes effect when the sender of an RSI call has to simultaneously transport one or more alarms. An example scenario is displayed in Figure 4.14. The Advanced and the Chunk Variants start at the same point of transportation. When an alarm occurs during the sending procedure of a window the alarm is sent directly.

In the chunk realization it is possible to add some part of current RSI transportation until the packet reaches the maximum packet size. One alarm chunk has a fixed size of 204 bytes. Since the maximum length is 1432 bytes there are still 1228 bytes left in that packet where 1224 bytes of RSI payload can be transported. Both payloads have an chunk header with the size of 4 bytes. The second alarm **AH** causes an increase of the 8 to 9 fragments that are necessary to transport the whole call. The question is in how many cases there is no extra fragment necessary to transport an alarm.

$$p_{no} = \frac{x_{al}}{x_{tot}} = \frac{len_{max} - len_{min} - len_{al}}{len_{max} - len_{min}} = \frac{1428 - 1 - 204}{1428 - 1} = 85{,}7\%$$

- p_{no}: Probability that a new alarm causes no new fragment.
- x_{al}: Cases where no new fragment is needed to transport an alarm.
- x_{tot}: All cases
- len_{max}: Maximum payload length of RSI.
- len_{min}: Minimum payload length of RSI.
- len_{al}: Length of alarm chunk.

In 86% of the cases where an alarm is sent simultaneously to an RSI call there is no extra fragment necessary to send.

If two alarms are in the send queue the alarm high is prioritized and then an RSI chunk is optionally appended. The chunk realization brings performance improvements but brings also complexity to the state machines.

The algorithm behind the designed sequences is the basis for the realization of the protocol. This is done by the state machines.

4.3. State Machine

Based on the sequence diagrams there are three new state machines for the Basic Variant: Initiator, Responder and (Lower Layer). Reduced versions of the Initiator and Responder state machines are illustrated in Figure 4.15 and 4.16. The complete versions are appended in Appendix C and D. These state machines are part of the PROFINET norm. The simplified visual presentation is sufficient to understand the functionality of RSI. The parts that are missing do not reduce the understanding of fragmentation of RSI. Device Access and Application Ready Notification are an example for extra features of RSI.

Application Ready Notification is a callback from the device to the controller after startup where it tells that is application is ready for cyclic data. It is the only call where the device is the initiator. RSI uses therefore an ACK PDU with a Notification bit **N** set to 1. The controller then initializes a call to bring the information. The Application Ready Notification mechanism is described in a separate state machine.

Device Access describes a mechanism where the connection will not be persistent. The call is addressed directly to the device.

4.3.1. Basic Variant

The state machines of the initiator and the responder are illustrated in Figure 4.15 and 4.16. Both are embedded in upper and lower state machines. The events that are described in the arrows are events from the upper and lower state machines which are defined in the tables below the machines.

The Initiator is activated with a *Connect.req*. This and the following requests are fragmented in the state Fragment Request **FREQ** and sent to the responder. The initiator starts with window size 1 and waits for an acknowledgement after the first fragment. The initiator stays in the **FREQ** state until it has received and acknowledegment for the last fragment of the request. This is either the case when the response is received or the Initator retransmits the last window and the responder sends an acknowledgment to tell that it is working on the request.

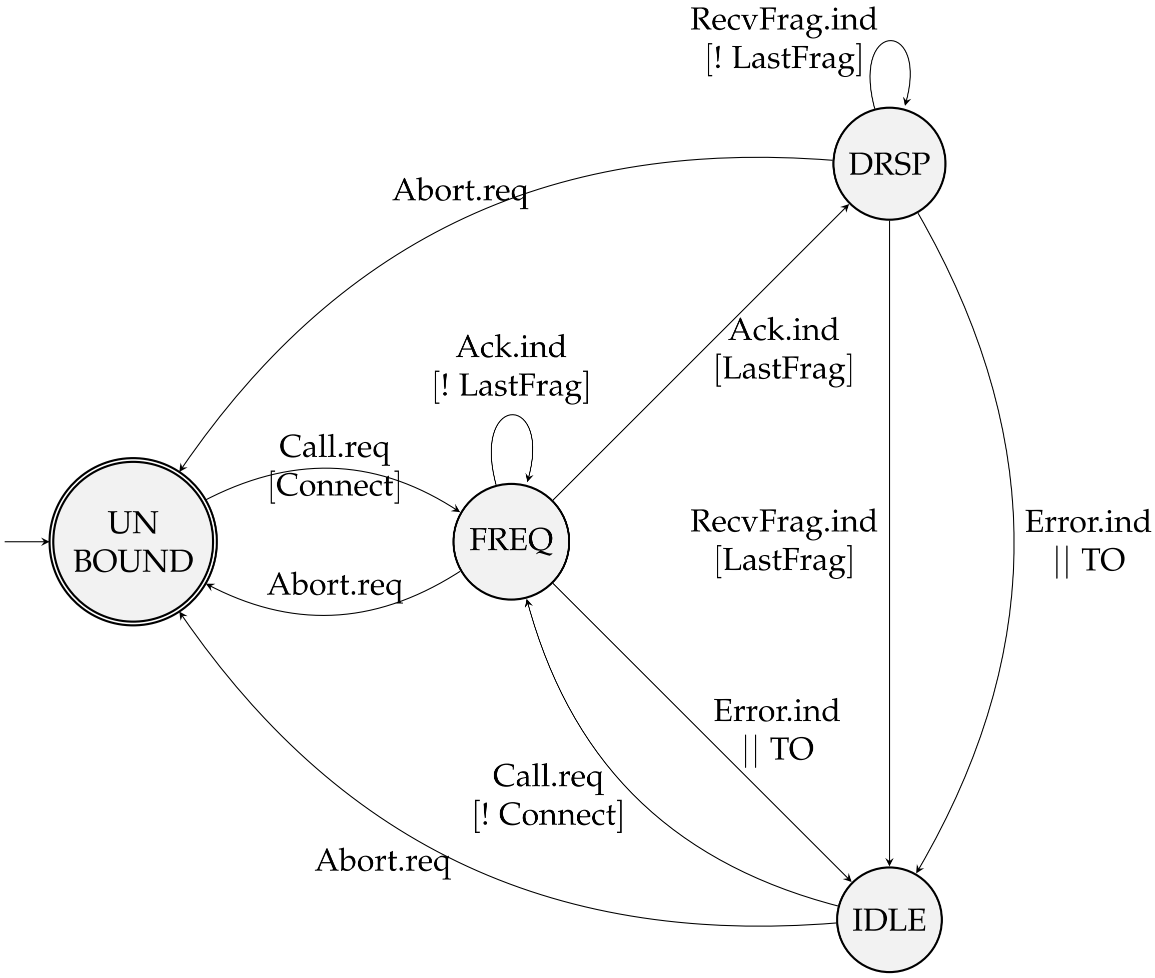


Figure 4.15.: Initiator state machine of the Basic Variant. The start state is **UNBOUND**. In **FREQ** the request is fragmented. In **DRSP** the response is defragmented. In the state **IDLE** the connection is established and a new request can be sent. (Pössler and Stefan, 2017a)

Table 4.2.: Events of Initiator state machine shown in Figure 4.15.

Name	Source	Description	Action
Call.req[*]	Upper	Demand to transport the call request with the opnum *.	SendFrag.req
SendFrag.req	Intern.	Send next unacknowledged window.	
Ack.ind[!LastF.]	Lower	Receive new AckSeqNum	SendFrag.req
Ack.ind[LastF.]	Lower	Receive new AckSeqNum [Last Frag].	
RecvFrag.ind [!LastF.] && TACK	Lower	Receive new SendSeqNum	SendAck.req
SendAck.req	Intern.	Send acknowledgment.	
RecvFrag.ind [!LastF.]	Lower	Receive new SendSeqNum [last Frag].	Call.cnf(+)
Abort.req	Upper	Abort connection.	Call.cnf(-)
Error.ind	Lower	Cancel current call by responder.	Call.cnf(-)
TO	Intern.	Call Timeout	Call.cnf(-)

Table 4.3.: Events of Responder state machine shown in Figure 4.16.

Name	Source	Description	Action
RecvFrag.ind [First Connect]	Lower	First fragment of connect indication	Start Timer
RecvFrag.ind [TACK]	Lower	Received last fragment from window	SendAck.req
RecvFrag.ind [LastFrag]	Lower	Received complete request.	Stop Timer
Call.rsp	Upper	Demand to transport the call response with the opnum *.	SendFrag.rsp
Ack.ind[!LastF.]	Lower	Receive new AckSeqNum	SendFrag.rsp
Ack.ind[LastF.]	Lower	New request acknowledges response implicit.	
SendError.req	Intern.	Sends error PDU.	Abort.cnf()
Abort.req	Upper	Abort connection	SendError.req
SendError.cnf	Lower	Error sent	Abort.cnf()
Error.ind	Lower	Cancel current call by initiator.	Call.cnf(-)
TO	Intern.	Connect request timeout	Call.cnf(-)

In the state Defragment Response **DRSP** the response is defragmented. When a window is received and the TACK flag is set the Initiator sends an ACK which tells the responder to continue to send the next window. When the response has been received completely the state changes to **IDLE**. The Initiator is ready for a new call.

The *Abort.req* is an abort of the connection that is triggerd from the upper layer and the *Error.ind* is a trigger from the Responder that it has finished the connection. The Abort.req can be triggered by the upper layer of the Initiator and the Responder. After finishing the connection both send an Error-PDU which triggers an Error.ind at the other communication partner. The Error-PDU will no be repeated if it is lost. In this case the the Initiator will be set to **UNBOUND** if a timeout occurs during an active call or during Abort.req by the user.

The Responder uses a request timeout which is active during a fragmented request in the state **DREQ**. When the request has been received completely

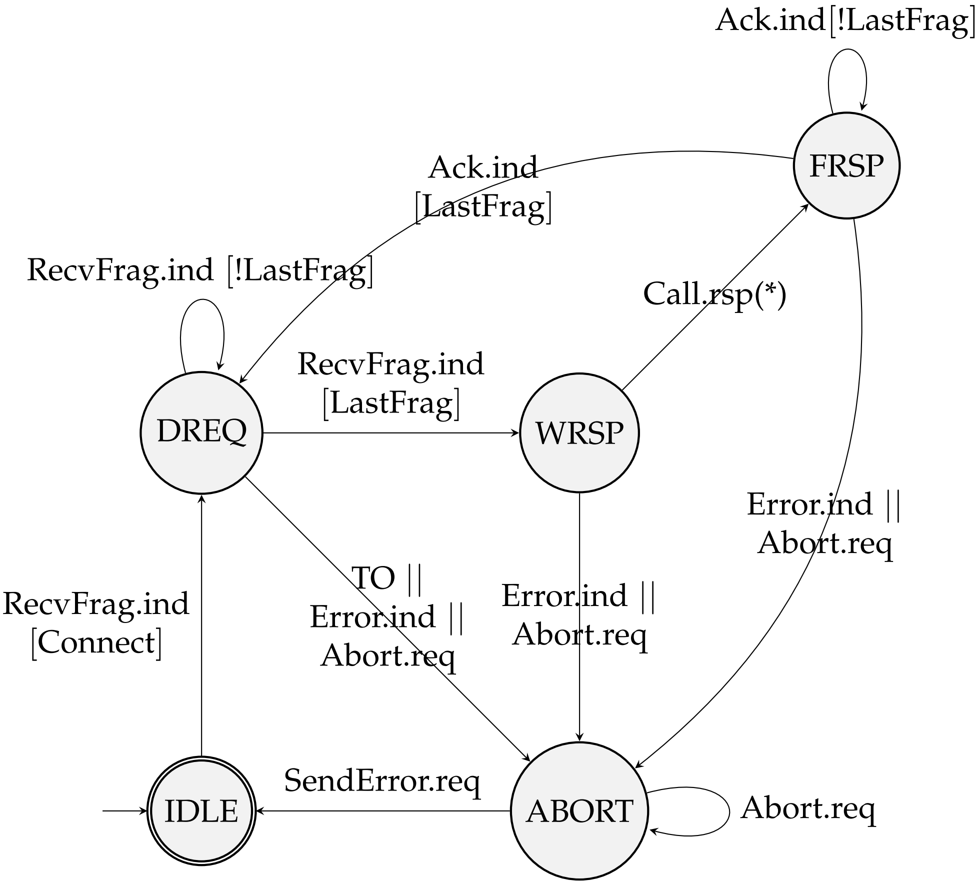


Figure 4.16.: Responder state machine of the Basic Variant. The initial state is **IDLE** where no connection is established. **DREQ** is the defragmentation of the request and **WRSP** waits for the response. The state **FRSP** sends the response and fragments ist. The state **ABORT** can be triggerd through timeout, incoming error PDU or an *Abort.req*. It aborts the connection and sends an Error PDU. (Pössler and Stefan, 2017b)

Table 4.4.: Events of an Alarm state machine shown in Figure 4.17.

Name	Source	Description	Action
Alarm.req	Upper	Alarm to send.	Insert in send queue. start alarm timer.
Alarm.ind	Lower	Incomming alarm	Send Ack-PDU.
Ack.ind	Lower	Update ACK Info and check if alarm is to send.	SendWindow or return to RSI.
TO	Intern	Alarm Timeout[not max]	SendWindow
TO_{Max}	Intern	Alarm Timeout [max]	
Error.ind	Lower	Error indication	Error.ind
Abort.req	Upper	Abort request	Abort.req

the responder delivers the request to its user and change the state to state to **WRSP**. The response from the upper layer is fragmented in the state **FRSP**. After an window has sent the responder waits for an acknowledgment from the Initiator. No acknowledgment will be sent for the last window so the responder will stay in **FRSP** until a new request will be received.

The initiator and the responder state machines are above the lower layer state machine which is described in Appendix F. It checks that the parameters are within the allowed boundaries and triggers acknowledgment-, RSI- and alarm indications. For the RSI indications it extracts the information if it is a connect request.

4.3.2. Advanced Variant

The Advanced Variant of RSI combines the RSI and alarm state machines in order to transport both in one transport window. To do this the state machines of the Basic Variant is extended by the one that is displayed in Figure 4.17. In the state **RSI** the alarm is not active. If an *Alarm.req* occurs the state changes to **ALARM SEND**. In this state the lower events trigger the Alarm state machine and not the RSI state machine.

The alarm that has to be sent will be inserted in the send array of RSI packets. It is inserted directly after the highest sent SendSeqNum. If there

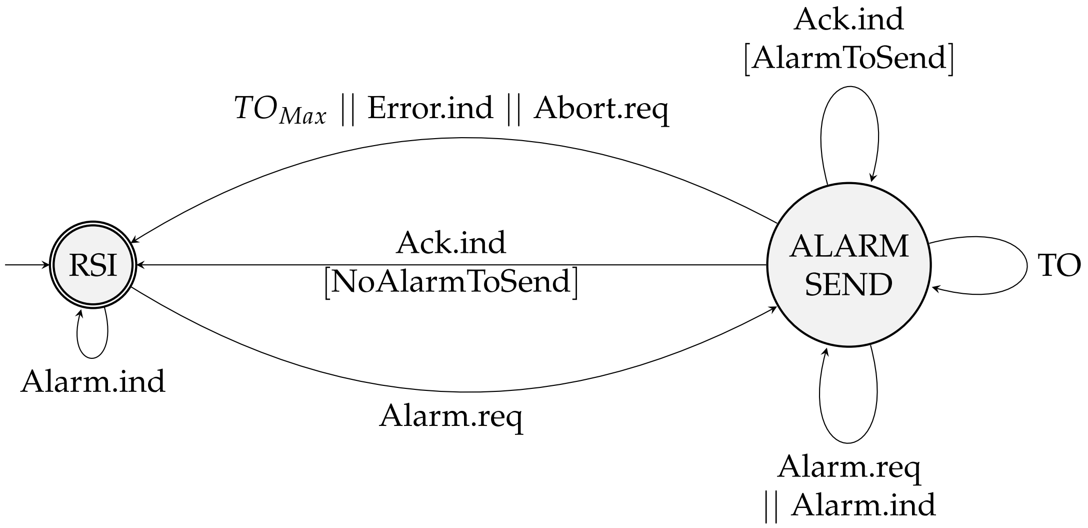


Figure 4.17.: Alarm state machine for Advanced Variant. It is an extension of the basic state machines and becomes active with an *Alarm.req*. *Alarm.ind* are always possible.

is still room in the window the next fragment is sent. The fragmentation timeout is stopped and the alarm timeout is started. If there is no free window the sender waits until the timeout occurs or an acknowledgment of the old window is received.

If another ***Alarm.req*** is executed the sender checks if an alarm with the same priority (low or high) is already in the queue. If there is none it is stored. Otherwise it is confirmed with a negative response. If an *Error.ind*, an *Abort.req* or a timeout occurs after the maximum time the state machines will be reset and the events are forwarded to the RSI state machine.

All events that are not defined in the state **ALARM SEND** are queued internally and after successful transmission of all alarms they are forwarded to the RSI state machine and are executed in the order they occurred. The complete state machine of the advanced variant is described in Appendix E.

4.3.3. Chunk Variant

For the the chunk realization the Advanced Variant is extended to put alarm and record PDUs in one packet. An alarm does not need a complete packet length for transportation and there is room to transport a part of a running call. The state machines for Initiator and Responder of the Advanced Variant has to be adapted, as well as the integration of the alarm for the Advanced Variant. Both variants divide the call in fragments at the beginning of the transportation. This is not possible if chunks were used. It is not certain how many fragments are there since alarms change the fragmentation and it is not deterministic when an alarm occurs.

The Basic Variant has been designed and is ready for the practical part. In the next chapter the implementation and the implementation environment is described.

5. Implementation

The theoretical design of the protocol in the previous chapter enables the integration of RSI in the PROFINET software stack. This work is done by a team with the size of circa five people plus system architects.

The PROFINET stack is developed in a Visual Studio project and is written in C. PCIOX is a Windows application, which includes the PROFINET stack and is made for testing purposes. PCIOX emulates IO-Devices and IO-Controllers which contain Slots and Subslots. The development architecture is illustrated in Figure 5.1.

The **Totally Integrated Automation (TIA)** portal is a software where the device configuration and network topology can be described. The device properties are then exported to a configuration file where the technical capabilities are stored. For the device another configuration file contains the inner architecture of slots, the address configuration. The INI-File of the controller includes its own address and slot configuration and the configuration of the devices that are controlled. (Popp, 2005, p. 258)

PCIOX controls the PCIe boards **SOC1** and **EB200P**. Both are Siemens development boards that are able to send and receive PROFINET Ethernet frames. In the test environment the SOC1 is used as the controller and the EB200P is used as the device. They are different in their architecture, both can be used as controller or device.

The EB200P card has four RJ45 connectors. Two for interacting and two for logging the incoming and out going frames. These two ports may be connected to **Wireshark** via another common network interfaces. It is running on a PC. How does it look like?

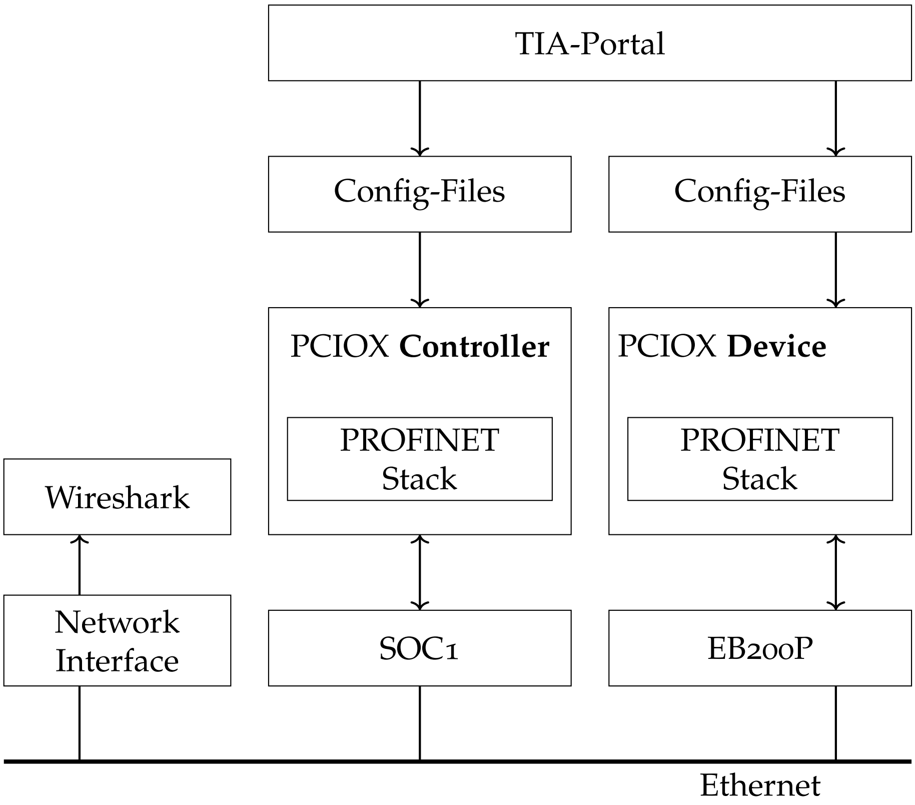


Figure 5.1.: Development environment. PCIOX is the PC application that includes the PROFINET stack and builds an IO-Device or IO-Controller in software. PCIOX controls the SOC1 and EB200P development boards. Config-Files are computed in the TIA-Portal and describes the device and controller. Wireshark captures the packets.

5.1. Test Interface

The PCIOX GUI is shown in Figure 5.2 where the connection between the devices has not been established. The controller has a description of the device in its own configuration which contains the slots, subslots and function descriptions. On the right side of the GUI there are tabs with information about PCIOX itself, the state of the connection and the possibility to write and read records after connection establishment. The controller initializes a connection to the device with the *Add* Request. After the connection phase has been finished the AR state changes to **IN DATA**, as shown in Figure 5.3.

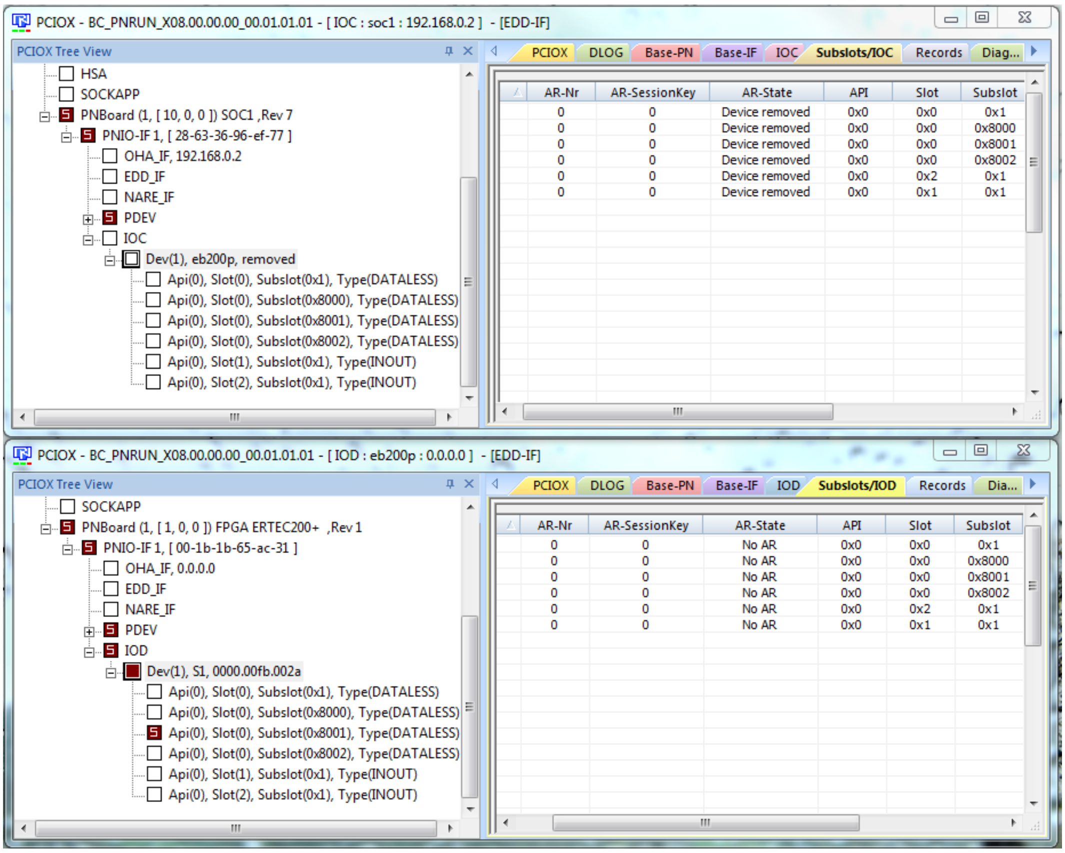
Beside of PCIOX another tool is used to monitor the communication.

5.2. Frame Observation

The fragmentation can not be displayed by PCIOX. Fragmentation and reassembly is done in the background. For debug purposes **Wireshark** (Combs, 2018) is used. Wireshark is a network protocol analyzer that can filter, reassemble and present network packets. PROFINET uses an adapted version with automatic highlighting and reassembly features. A snapshot of the frames of the connection phase is displayed in Figure 5.4. All frames are smaller than the maximum ***VarPartLen***. No fragmentation is used. The fragment information is decoded in the ***FOpnumOffset*** field which includes ***Call Sequence***, ***Opnum*** and ***Offset***.

In Figure 5.5 a fragmented *Write* request is performed which consists of 10 fragments. The window size is two. After two fragments has been sent the device sends an *ACK-RTA, Application Ready Notification*. The acknowledgment PDU may be used additionally to tell the controller that the application is ready. The modified Wireshark also has the functionality to reassemble fragmented calls. By clicking on the last frame the whole call is readable.

The communication can be recorded and so evaluated.



PCIOX - BC_PNRUN_X08.00.00.00_00.01.01.01 - [IOC : soc1 : 192.168.0.2] - [EDD-IF]

- □ HSA
- □ SOCKAPP
- PNBoard (1, [10, 0, 0]) SOC1 ,Rev 7
  - PNIO-IF 1, [28-63-36-96-ef-77]
    - □ OHA_IF, 192.168.0.2
    - □ EDD_IF
    - □ NARE_IF
    - PDEV
    - □ IOC
      - □ Dev(1), eb200p, removed
        - □ Api(0), Slot(0), Subslot(0x1), Type(DATALESS)
        - □ Api(0), Slot(0), Subslot(0x8000), Type(DATALESS)
        - □ Api(0), Slot(0), Subslot(0x8001), Type(DATALESS)
        - □ Api(0), Slot(0), Subslot(0x8002), Type(DATALESS)
        - □ Api(0), Slot(1), Subslot(0x1), Type(INOUT)
        - □ Api(0), Slot(2), Subslot(0x1), Type(INOUT)

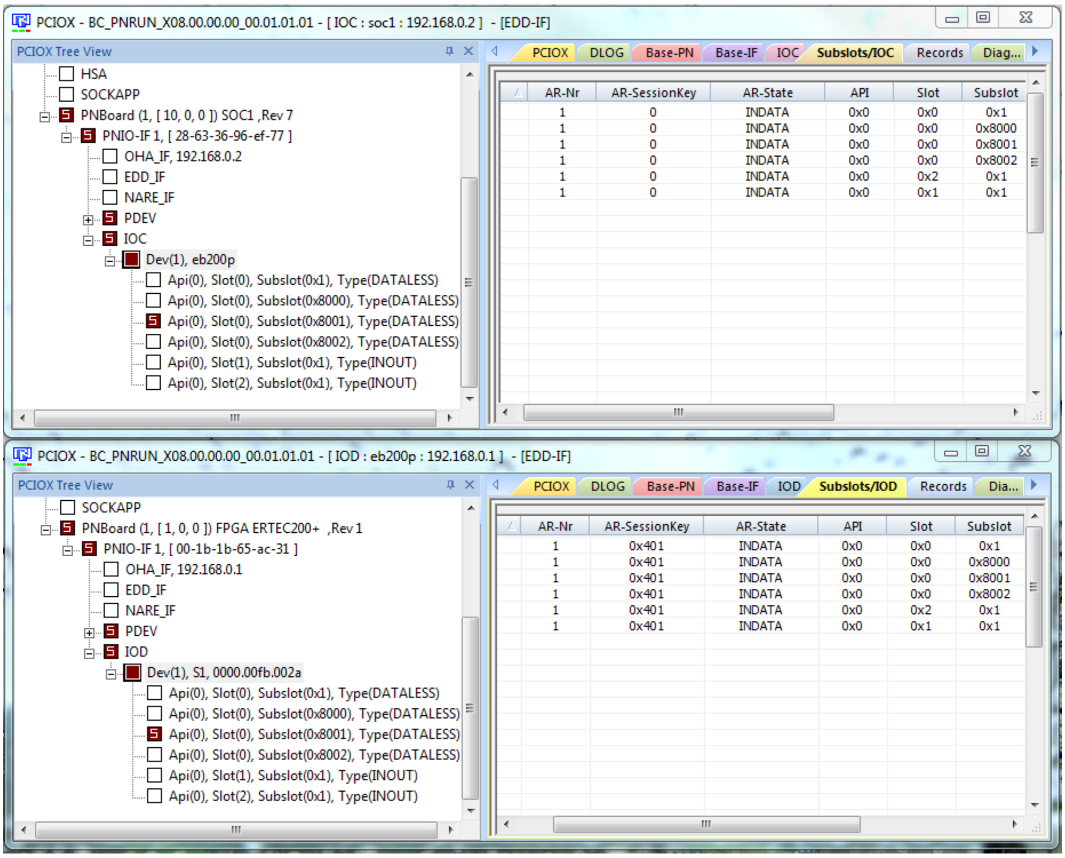
AR-Nr	AR-SessionKey	AR-State	API	Slot	Subslot
0	0	Device removed	0x0	0x0	0x1
0	0	Device removed	0x0	0x0	0x8000
0	0	Device removed	0x0	0x0	0x8001
0	0	Device removed	0x0	0x0	0x8002
0	0	Device removed	0x0	0x2	0x1
0	0	Device removed	0x0	0x1	0x1

PCIOX - BC_PNRUN_X08.00.00.00_00.01.01.01 - [IOD : eb200p : 0.0.0.0] - [EDD-IF]

- □ SOCKAPP
- PNBoard (1, [1, 0, 0]) FPGA ERTEC200+ ,Rev 1
  - PNIO-IF 1, [00-1b-1b-65-ac-31]
    - □ OHA_IF, 0.0.0.0
    - □ EDD_IF
    - □ NARE_IF
    - PDEV
    - IOD
      - Dev(1), S1, 0000.00fb.002a
        - □ Api(0), Slot(0), Subslot(0x1), Type(DATALESS)
        - □ Api(0), Slot(0), Subslot(0x8000), Type(DATALESS)
        - Api(0), Slot(0), Subslot(0x8001), Type(DATALESS)
        - □ Api(0), Slot(0), Subslot(0x8002), Type(DATALESS)
        - □ Api(0), Slot(1), Subslot(0x1), Type(INOUT)
        - □ Api(0), Slot(2), Subslot(0x1), Type(INOUT)

AR-Nr	AR-SessionKey	AR-State	API	Slot	Subslot
0	0	No AR	0x0	0x0	0x1
0	0	No AR	0x0	0x0	0x8000
0	0	No AR	0x0	0x0	0x8001
0	0	No AR	0x0	0x0	0x8002
0	0	No AR	0x0	0x2	0x1
0	0	No AR	0x0	0x1	0x1

Figure 5.2.: PCIOX GUI of controller (top) and device (bottom) with no active Application Relation (No AR). The structure of the Interface is illustrated on the left in the tree view. The controller is configured to control the device 1 with the listed slots and subslots. If not connected the controller displays *Device Removed* and the device displays NO AR.



PCIOX - BC_PNRUN_X08.00.00.00_00.01.01.01 - [IOC : soc1 : 192.168.0.2] - [EDD-IF]

```
HSA
SOCKAPP
PNBoard (1, [ 10, 0, 0 ]) SOC1 ,Rev 7
  PNIO-IF 1, [ 28-63-36-96-ef-77 ]
    OHA_IF, 192.168.0.2
    EDD_IF
    NARE_IF
    PDEV
    IOC
      Dev(1), eb200p
        Api(0), Slot(0), Subslot(0x1), Type(DATALESS)
        Api(0), Slot(0), Subslot(0x8000), Type(DATALESS)
        Api(0), Slot(0), Subslot(0x8001), Type(DATALESS)
        Api(0), Slot(0), Subslot(0x8002), Type(DATALESS)
        Api(0), Slot(1), Subslot(0x1), Type(INOUT)
        Api(0), Slot(2), Subslot(0x1), Type(INOUT)
```

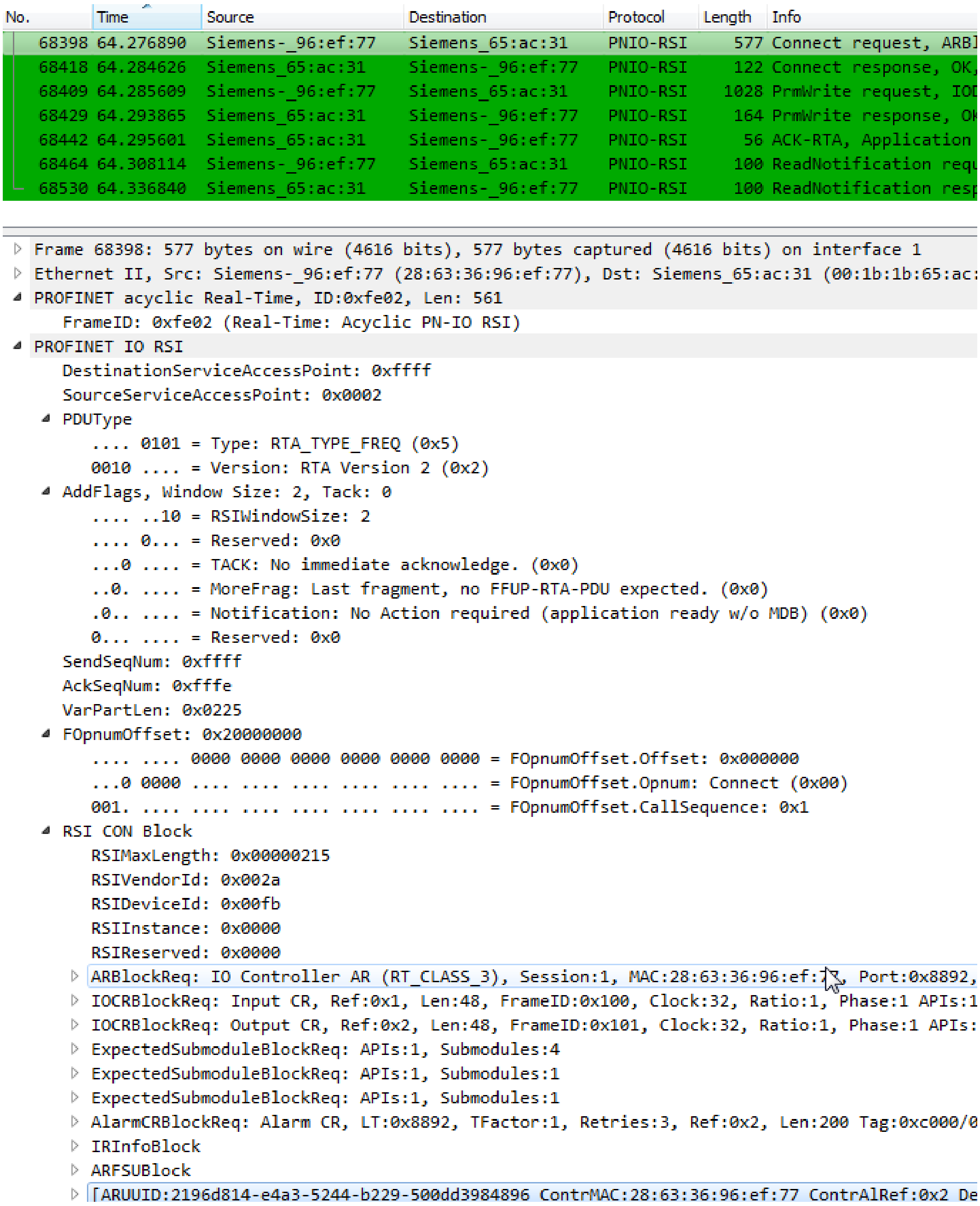
AR-Nr	AR-SessionKey	AR-State	API	Slot	Subslot
1	0	INDATA	0x0	0x0	0x1
1	0	INDATA	0x0	0x0	0x8000
1	0	INDATA	0x0	0x0	0x8001
1	0	INDATA	0x0	0x0	0x8002
1	0	INDATA	0x0	0x2	0x1
1	0	INDATA	0x0	0x1	0x1

PCIOX - BC_PNRUN_X08.00.00.00_00.01.01.01 - [IOD : eb200p : 192.168.0.1] - [EDD-IF]

```
SOCKAPP
PNBoard (1, [ 1, 0, 0 ]) FPGA ERTEC200+ ,Rev 1
  PNIO-IF 1, [ 00-1b-1b-65-ac-31 ]
    OHA_IF, 192.168.0.1
    EDD_IF
    NARE_IF
    PDEV
    IOD
      Dev(1), S1, 0000.00fb.002a
        Api(0), Slot(0), Subslot(0x1), Type(DATALESS)
        Api(0), Slot(0), Subslot(0x8000), Type(DATALESS)
        Api(0), Slot(0), Subslot(0x8001), Type(DATALESS)
        Api(0), Slot(0), Subslot(0x8002), Type(DATALESS)
        Api(0), Slot(1), Subslot(0x1), Type(INOUT)
        Api(0), Slot(2), Subslot(0x1), Type(INOUT)
```

AR-Nr	AR-SessionKey	AR-State	API	Slot	Subslot
1	0x401	INDATA	0x0	0x0	0x1
1	0x401	INDATA	0x0	0x0	0x8000
1	0x401	INDATA	0x0	0x0	0x8001
1	0x401	INDATA	0x0	0x0	0x8002
1	0x401	INDATA	0x0	0x2	0x1
1	0x401	INDATA	0x0	0x1	0x1

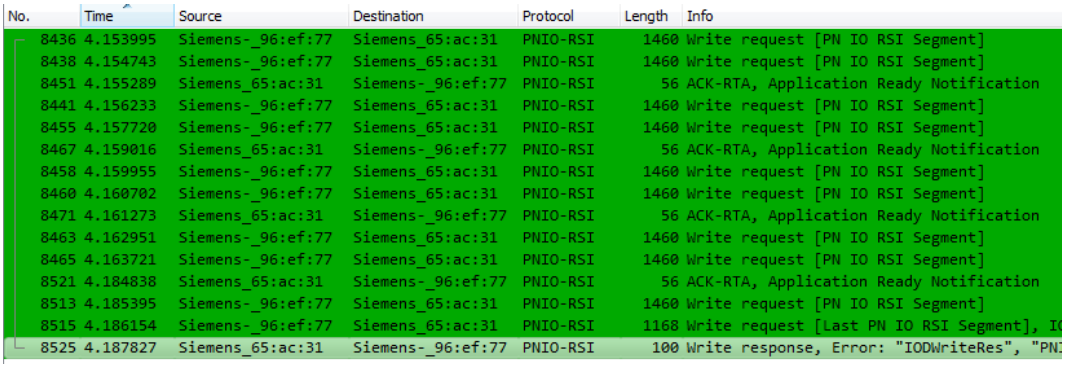
Figure 5.3.: Snapshot of PCIOX GUI of controller (top) and device (bottom) with an active Application Relation (AR). The AR-State has changed on both sides to INDATA and the controller is able to send read and write records.



```
No.       Time         Source             Destination        Protocol   Length Info
   68398 64.276890  Siemens-_96:ef:77  Siemens_65:ac:31   PNIO-RSI      577 Connect request, ARBl
   68418 64.284626  Siemens_65:ac:31   Siemens-_96:ef:77  PNIO-RSI      122 Connect response, OK,
   68409 64.285609  Siemens-_96:ef:77  Siemens_65:ac:31   PNIO-RSI     1028 PrmWrite request, IOD
   68429 64.293865  Siemens_65:ac:31   Siemens-_96:ef:77  PNIO-RSI      164 PrmWrite response, OK
   68442 64.295601  Siemens_65:ac:31   Siemens-_96:ef:77  PNIO-RSI       56 ACK-RTA, Application
   68464 64.308114  Siemens-_96:ef:77  Siemens_65:ac:31   PNIO-RSI      100 ReadNotification requ
   68530 64.336840  Siemens_65:ac:31   Siemens-_96:ef:77  PNIO-RSI      100 ReadNotification resp

▷ Frame 68398: 577 bytes on wire (4616 bits), 577 bytes captured (4616 bits) on interface 1
▷ Ethernet II, Src: Siemens-_96:ef:77 (28:63:36:96:ef:77), Dst: Siemens_65:ac:31 (00:1b:1b:65:ac:
◢ PROFINET acyclic Real-Time, ID:0xfe02, Len: 561
    FrameID: 0xfe02 (Real-Time: Acyclic PN-IO RSI)
◢ PROFINET IO RSI
    DestinationServiceAccessPoint: 0xffff
    SourceServiceAccessPoint: 0x0002
  ◢ PDUType
      .... 0101 = Type: RTA_TYPE_FREQ (0x5)
      0010 .... = Version: RTA Version 2 (0x2)
  ◢ AddFlags, Window Size: 2, Tack: 0
      .... ..10 = RSIWindowSize: 2
      .... 0... = Reserved: 0x0
      ...0 .... = TACK: No immediate acknowledge. (0x0)
      ..0. .... = MoreFrag: Last fragment, no FFUP-RTA-PDU expected. (0x0)
      .0.. .... = Notification: No Action required (application ready w/o MDB) (0x0)
      0... .... = Reserved: 0x0
    SendSeqNum: 0xffff
    AckSeqNum: 0xfffe
    VarPartLen: 0x0225
  ◢ FOpnumOffset: 0x20000000
      .... .... 0000 0000 0000 0000 0000 0000 = FOpnumOffset.Offset: 0x000000
      ...0 0000 .... .... .... .... .... .... = FOpnumOffset.Opnum: Connect (0x00)
      001. .... .... .... .... .... .... .... = FOpnumOffset.CallSequence: 0x1
  ◢ RSI CON Block
      RSIMaxLength: 0x00000215
      RSIVendorId: 0x002a
      RSIDeviceId: 0x00fb
      RSIInstance: 0x0000
      RSIReserved: 0x0000
    ▷ ARBlockReq: IO Controller AR (RT_CLASS_3), Session:1, MAC:28:63:36:96:ef:77, Port:0x8892,
    ▷ IOCRBlockReq: Input CR, Ref:0x1, Len:48, FrameID:0x100, Clock:32, Ratio:1, Phase:1 APIs:1
    ▷ IOCRBlockReq: Output CR, Ref:0x2, Len:48, FrameID:0x101, Clock:32, Ratio:1, Phase:1 APIs:
    ▷ ExpectedSubmoduleBlockReq: APIs:1, Submodules:4
    ▷ ExpectedSubmoduleBlockReq: APIs:1, Submodules:1
    ▷ ExpectedSubmoduleBlockReq: APIs:1, Submodules:1
    ▷ AlarmCRBlockReq: Alarm CR, LT:0x8892, TFactor:1, Retries:3, Ref:0x2, Len:200 Tag:0xc000/0
    ▷ IRInfoBlock
    ▷ ARFSUBlock
    ▷ [ARUUID:2196d814-e4a3-5244-b229-500dd3984896 ContrMAC:28:63:36:96:ef:77 ContrAlRef:0x2 De
```

Figure 5.4.: Snapshot of Wireshark from the connection phase. The frames sent and received frames are displayed in the top part and the decoded information of the first packet is displayed in the bottom part.



No.	Time	Source	Destination	Protocol	Length	Info
8436	4.153995	Siemens-_96:ef:77	Siemens_65:ac:31	PNIO-RSI	1460	Write request [PN IO RSI Segment]
8438	4.154743	Siemens-_96:ef:77	Siemens_65:ac:31	PNIO-RSI	1460	Write request [PN IO RSI Segment]
8451	4.155289	Siemens_65:ac:31	Siemens-_96:ef:77	PNIO-RSI	56	ACK-RTA, Application Ready Notification
8441	4.156233	Siemens-_96:ef:77	Siemens_65:ac:31	PNIO-RSI	1460	Write request [PN IO RSI Segment]
8455	4.157720	Siemens-_96:ef:77	Siemens_65:ac:31	PNIO-RSI	1460	Write request [PN IO RSI Segment]
8467	4.159016	Siemens_65:ac:31	Siemens-_96:ef:77	PNIO-RSI	56	ACK-RTA, Application Ready Notification
8458	4.159955	Siemens-_96:ef:77	Siemens_65:ac:31	PNIO-RSI	1460	Write request [PN IO RSI Segment]
8460	4.160702	Siemens-_96:ef:77	Siemens_65:ac:31	PNIO-RSI	1460	Write request [PN IO RSI Segment]
8471	4.161273	Siemens_65:ac:31	Siemens-_96:ef:77	PNIO-RSI	56	ACK-RTA, Application Ready Notification
8463	4.162951	Siemens-_96:ef:77	Siemens_65:ac:31	PNIO-RSI	1460	Write request [PN IO RSI Segment]
8465	4.163721	Siemens-_96:ef:77	Siemens_65:ac:31	PNIO-RSI	1460	Write request [PN IO RSI Segment]
8521	4.184838	Siemens_65:ac:31	Siemens-_96:ef:77	PNIO-RSI	56	ACK-RTA, Application Ready Notification
8513	4.185395	Siemens-_96:ef:77	Siemens_65:ac:31	PNIO-RSI	1460	Write request [PN IO RSI Segment]
8515	4.186154	Siemens-_96:ef:77	Siemens_65:ac:31	PNIO-RSI	1168	Write request [Last PN IO RSI Segment], IO
8525	4.187827	Siemens_65:ac:31	Siemens-_96:ef:77	PNIO-RSI	100	Write response, Error: "IODWriteRes", "PNI

Figure 5.5.: Snapshot of Wireshark from a fragmented request with 10 fragments. Window size is 2.

5.3. Time

The time to start a communication relation can be compared between RSI and CLRPC. Both have a similar start up behavior. RSI needs three calls with the following ***Opnums***: *Connect*, *PrmWrite* and *ReadNotification* to reach the state **INDATA**. CLRPC needs an additional call: *PrmEnd*.

From the start of the connection to the state **INDATA** RSI needs 3,78s and CLRPC 3,90s. RSI is 0,13 s faster than RPC for startup, which includes allocation of memory and is not only linked to the fragmentation process. By now CLRPC is not removable and so the RSI variant also allocates the buffer for CLRPC. In this test all calls are not fragmented.

The transportation time of a fragmented call is measured. The measurements are made with the Light Variant of PCIOX, where the computations are made in Windows. It is done through an internal time logging function since it has a larger accuracy than the network logging process. The time is measured two times: Send point of the first fragment of a call and the point when the last fragment of a call is sent. The results of the comparison are displayed in Figure 5.6. The figure displays the averaged transport duration of a call with the length of 25 kBytes. The averaged value of CLRPC is 54μs and RSI has an average transportation time of 43μs. The upper border time of both variants is the same. The lower border differs. In RSI some transportations need only 23μs. In CLRPC all transportations take 46μs

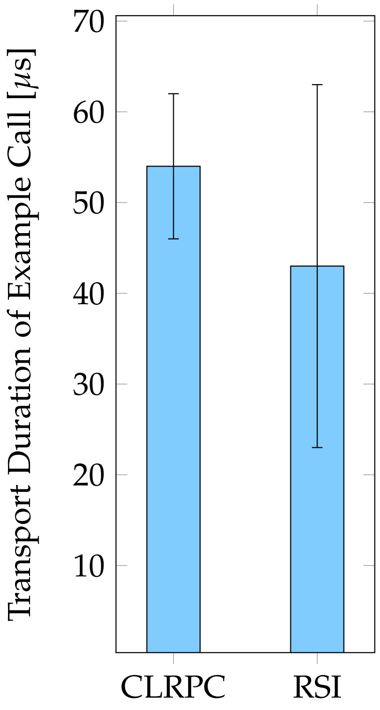


Figure 5.6.: Comparison of Duration in μs of a fragmented call between CLRPC and RSI. The highest 10% and the lowest 10% are not included of all measured cases are removed.

or longer. RSI calls are not slower than CLRPC calls. On average they are faster.

Besides the time analysis the need of memory and the code size are limiting factors.

5.4. Memory and Code Size

The code size and the used runtime memory of RTA v1 and RTA v2 are analyzed.

The **memory** includes stack and heap. Three different constellations are analyzed: CLRPC only, CLRPC and RSI or RSI only. Controllers always have to support CLRPC and optionally RSI. So the removal of CLRPC from the PROFINET stack may only be a memory benefit for CLRPC. If RSI and CLRPC are used the memory use is 4 Kbyte higher than it is without RSI. At this point CLRPC cannot be completely removed from the build so it is impossible to make a comparison of the real memory benefit when CLRPC is not used.

The results of the **code size** analysis are displayed in Figure 5.7. The analysis is done for the Ertec200p with an OpenBSD stack. The component CLRPC is the component where the RPC calls are performed. SOCK and TCIP are there for the IP and UDP/TCP stacks. These stacks are not necessary in v2 of RTA. The implementation may reduce the codesize by about 321 kBytes.

The implementation especially the testing of RSI is in progress and will be continued so that more analyses of RSI can be made. There is still work to do and the next big step is to have a fair construction with integrated RSI.

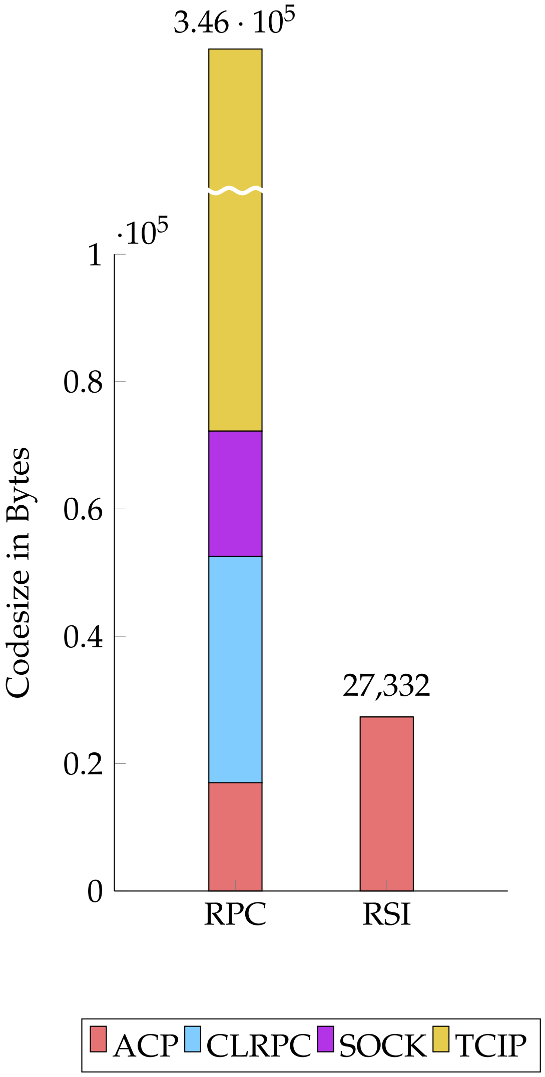


Figure 5.7.: Code size of the components in bits of the compile variant OpenBSD for ERTEC200p. Displayed are the involved components for the variant with records over RPC and of the variant with records over RSI without the other components.

6. Conclusion

Records for the parameterization have been sent over IP with the PROFINET component Connectionless Remote Procedure Calls **CLRPC**. The records that are sent with Remote Procedure Calls **RPC** over **IPv4** are ported to the Realtime Acyclic transport protocol **RTA**. RTA uses no extra network protocol like IP. In v1 RTA is used to send alarms. In v2 this is extended with the Remote Service Interface **RSI** to send records like RPC which may have a size of up to 16 MBytes.

RTA has been fragmented in order to be ready for RSI which should guarantee in order transportation and frame drop detection. There are three variants: Basic, Advanced and Chunk. The Basic Variant has been implemented. The fragment information is fix coded in the RSI payload. The communication is triggered and controlled by the controller. The device needs only a timer between the very first and the second fragment of a communication.

The other variants, Advanced and Chunk are designed in theory. Both could save receive resources because they use a common communication window for records and alarms. The chunk variant would even reduce the possibility that one protocol may break another because alarms and records could be sent within one packet. The benefit of chunks is limited to cases where an alarm occurs during a transportation of a fragmented call on the sender's side.

In comparison to CLRPC the resources are not extended. RSI transports a fragmented call 15% faster on average than CLRPC and the phase to establish an active Application Relation is 0,13s faster for the RSI implementation even though it is only a temporary result because CLRPC and the IP components are not removable so far.

RSI has solved the problem of call classes of CLRPC. The user had to determine whether the transportation of a call is guaranteed or it should only be executed once. In RSI a call is executed only once by default and the transportation is guaranteed. The device saves the data until a new call is received in the own buffer and in case of retransmission the response is sent back without executing the request.

With RSI the record services are ported to the Layer 2 of the OSI-Model like the realtime communication before. In the future PROFINT will be using Time Sensitive Network **TSN**, an open standard for real time communication with the goal of protecting the transportation undisturbed. For the TSN realization no IP stack will be used. RSI is an important base to port PROFINET to TSN and to transport large calls.

Furthermore RSI is capable enough to build a new communication standard in the PROFINET protocol suite for future-proofness. The door has been opened for new product lines of devices with additional feature and interface possibilities and beyond new product lines where the components connected to IP are removed completely.

Bibliography

Bormann, Alexander and Ingo Hilgenkamp (2006). *Industrielle Netze*. Hüthig. ISBN: 3778529501 (cit. on p. 3).

CAE Specification (1997). *DCE1.1: Remote Procedure Call*. The Open Group (cit. on pp. 29–34, 51, 53).

CCITT (1988). *SERIES X: DATA COMMUNICATION NETWORKS: OPEN SYSTEMS INTERCONNECTION (OSI) – MODEL AND NOTATION, SERVICE DEFINITION*. International Telecomunication Union. URL: https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.200-198811-S!!PDF-E&type=items (visited on 08/18/2017) (cit. on p. 35).

Combs, Gerald (2018). *About Wireshark*. URL: https://www.wireshark.org/ (visited on 02/05/2018) (cit. on p. 89).

Henning, Carl (2015). *Tech Tip: What Is PROFINET DCP?* URL: http://profinews.com/2015/07/tech-tip-what-is-profinet-dcp/ (visited on 12/30/2017) (cit. on p. 5).

Henning, Carl (2017). *Integrating TSN into PROFINET*. URL: http://profinews.com/2017/05/integrating-tsn-into-profinet/ (visited on 11/20/2017) (cit. on p. 1).

IANA (2017[a]). *Assigned Internet Protocol Numbers*. URL: https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml (visited on 08/07/2017) (cit. on p. 14).

IANA (2017[b]). *Service Name and Transport Protocol Port Number Registry*. URL: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml (visited on 11/07/2017) (cit. on p. 11).

IEC 61158-6-10 (2015). *Industrial Communication Networks – Fieldbus Specifications. Part 6–10: Application layer protocol specification – Type 10 elements*. International Eletrotechnical Commission (cit. on p. 55).

Popp, Manfred (2005). *Das PROFINET IO-Buch*. Hüthig. ISBN: 3778529668 (cit. on pp. 3–6, 87).

Pössler, Thomas (2018). *LMPM (Lower Layer) State Machine*. Tech. rep. Siemens AG (cit. on p. 135).

Pössler, Thomas and Matthias Stefan (2017a). *RSI Initiator State Machine*. Tech. rep. State Machine is designed by Pössler which is based on a version from me. Siemens AG (cit. on pp. 79, 117).

Pössler, Thomas and Matthias Stefan (2017b). *RSI Responder State Machine*. Tech. rep. State Machine is designed by Pössler which is based on a version from me. Siemens AG (cit. on pp. 82, 123).

PROFIBUS & PROFINET International (2016). *PROFINET System Description*. URL: https://www.profibus.com/download/profinet-technology-and-application-system-description/ (visited on 08/03/2017) (cit. on p. 1).

PROFIBUS & PROFINET International (2017[a]). *Case studies by technology and business*. URL: http://www.profibus.com/index.php?id=5013&pxdprofibusfilter_technology=1 (visited on 08/03/2017) (cit. on p. 1).

PROFIBUS & PROFINET International (2017[b]). *PROFINET - the leading Industrial Ethernet Standard*. URL: http://www.profibus.com/technology/profinet/overview/ (visited on 08/03/2017) (cit. on p. 1).

RFC 1700 (1994). *Assigned Numbers*. Network Working Group. URL: https://tools.ietf.org/pdf/rfc1700.pdf (visited on 08/08/2017) (cit. on p. 16).

RFC 1791 (1999). *TCP And UDP Over IPX Networks With Fixed Path MTU*. Network Working Group. URL: https://tools.ietf.org/pdf/rfc1791.pdf (visited on 11/30/2017) (cit. on p. 36).

RFC 2460 (1998). *Internet Protocol, Version 6 (IPv6)*. Network Working Group. URL: https://tools.ietf.org/pdf/rfc2460.pdf (visited on 08/07/2017) (cit. on pp. 16–19, 53).

RFC 2616 (1999). *Hypertext Transfer Protocol – HTTP/1.1*. Network Working Group. URL: https://tools.ietf.org/pdf/rfc2616.pdf (visited on 11/30/2017) (cit. on p. 12).

RFC 3286 (2002). *An Introduction to the Stream Control Transmission Protocol (SCTP)*. Network Working Group. URL: https://tools.ietf.org/pdf/rfc3286.pdf (visited on 08/16/2017) (cit. on p. 24).

RFC 4960 (2007). *Stream Control Transmission Protocol*. Network Working Group. URL: https://tools.ietf.org/pdf/rfc4960.pdf (visited on 08/16/2017) (cit. on pp. 25–29, 47, 51, 53).

RFC 7296 (2014). *Internet Key Exchange Protocol Version 2 (IKEv2)*. Internet Engineering Task Force. URL: https://tools.ietf.org/pdf/rfc7296.pdf (visited on 08/08/2017) (cit. on pp. 19–22, 47).

RFC 7383 (2014). *Internet Key Exchange Protocol Version 2 (IKEv2) Message Fragmentation*. Internet Engineering Task Force. URL: https://tools.ietf.org/pdf/rfc7383.pdf (visited on 08/08/2017) (cit. on pp. 20, 22–24).

RFC 760 (1981). *Internet Protocol*. Information Sciences Institute, University of Southen California. URL: https://tools.ietf.org/pdf/rfc760.pdf (visited on 07/03/2017) (cit. on pp. 12–15).

RFC 791 (1981). *IP - ARPA Internet Program*. Information Sciences Institute, University of Southen California. URL: https://tools.ietf.org/pdf/rfc791 (visited on 08/07/2017) (cit. on pp. 14, 53).

RFC 793 (1980). *Transmission Control Protocol*. Information Sciences Institute, University of Southen California. URL: https://tools.ietf.org/html/rfc793 (visited on 06/28/2017) (cit. on pp. 11, 47, 48, 51).

Siemens AG (2017). *Totally Integrated Automation Portal*. URL: https://www.siemens.com/global/de/home/produkte/automatisierung/industrie-software/automatisierungs-software/tia-portal.html (visited on 11/29/2017) (cit. on p. 5).

Appendix

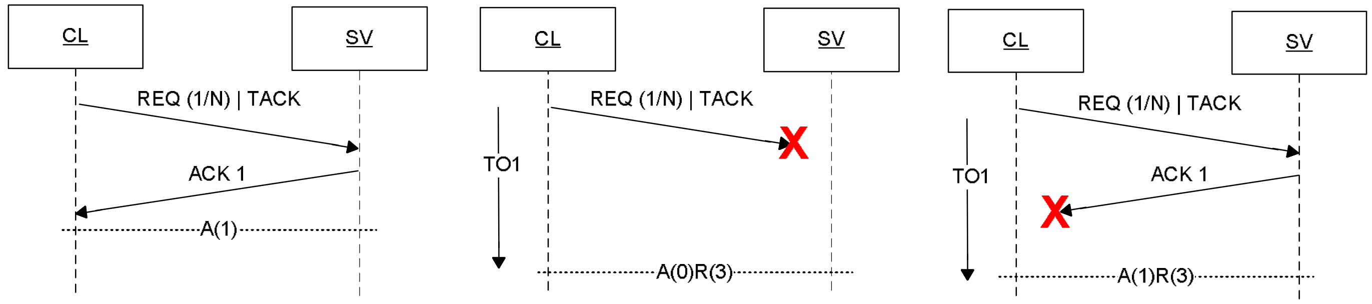
Appendix A.

RTA Fragmentation Variant Basic - Sequence

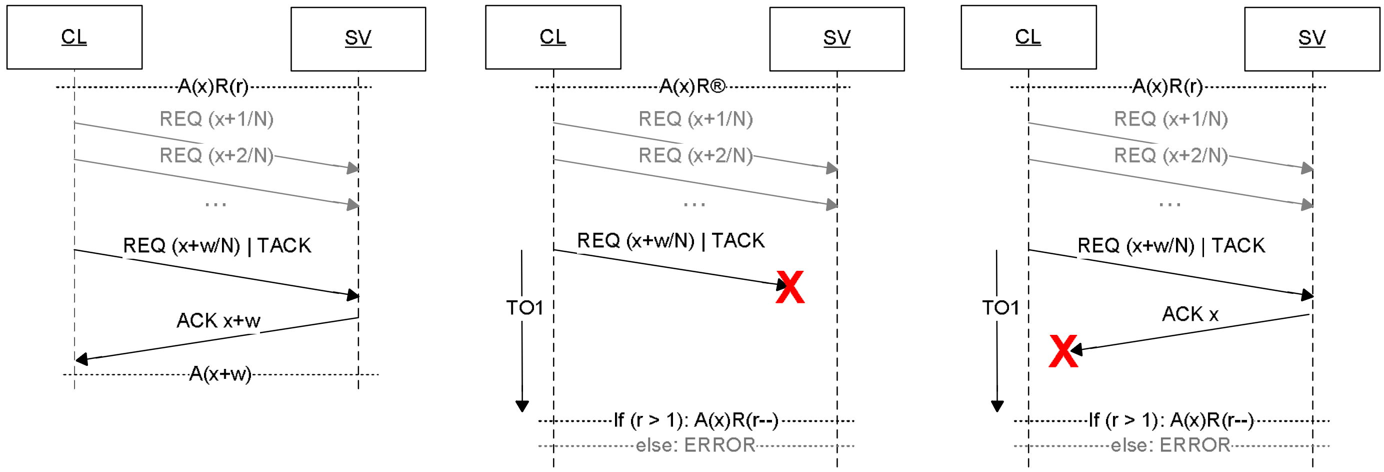
A(x): x fragments acknowledged of REQ
A(x)R(r): x fragments acknowledged of REQ with failure -> r retries
W/R(r): Server is Working. Controller waits for Read
RSP(x)/R(r): Response received and r retries until ERROR

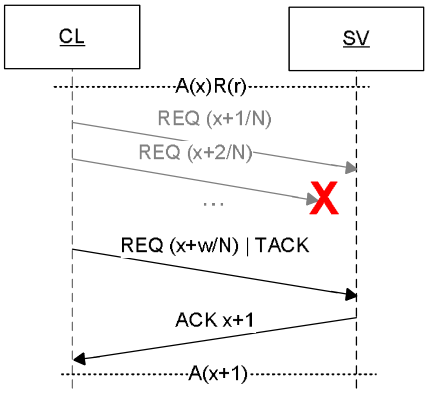
N: Length of Request
M: Length of Response

START



Repeat Acknowledgment
of x and have r retries
(including this)

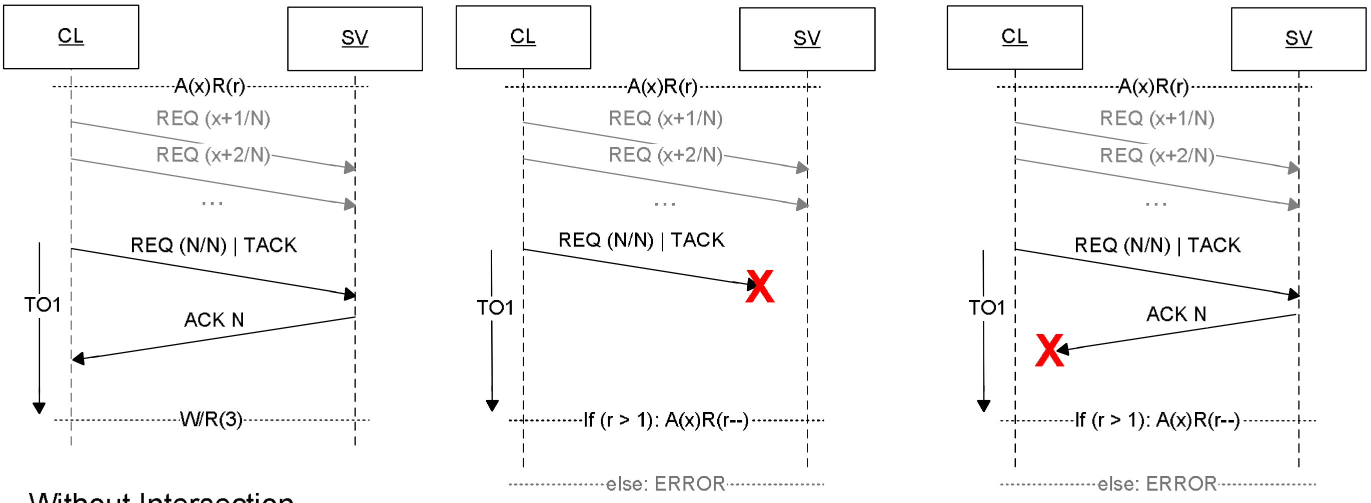




A(Last Received in order)

A(x)R(r)

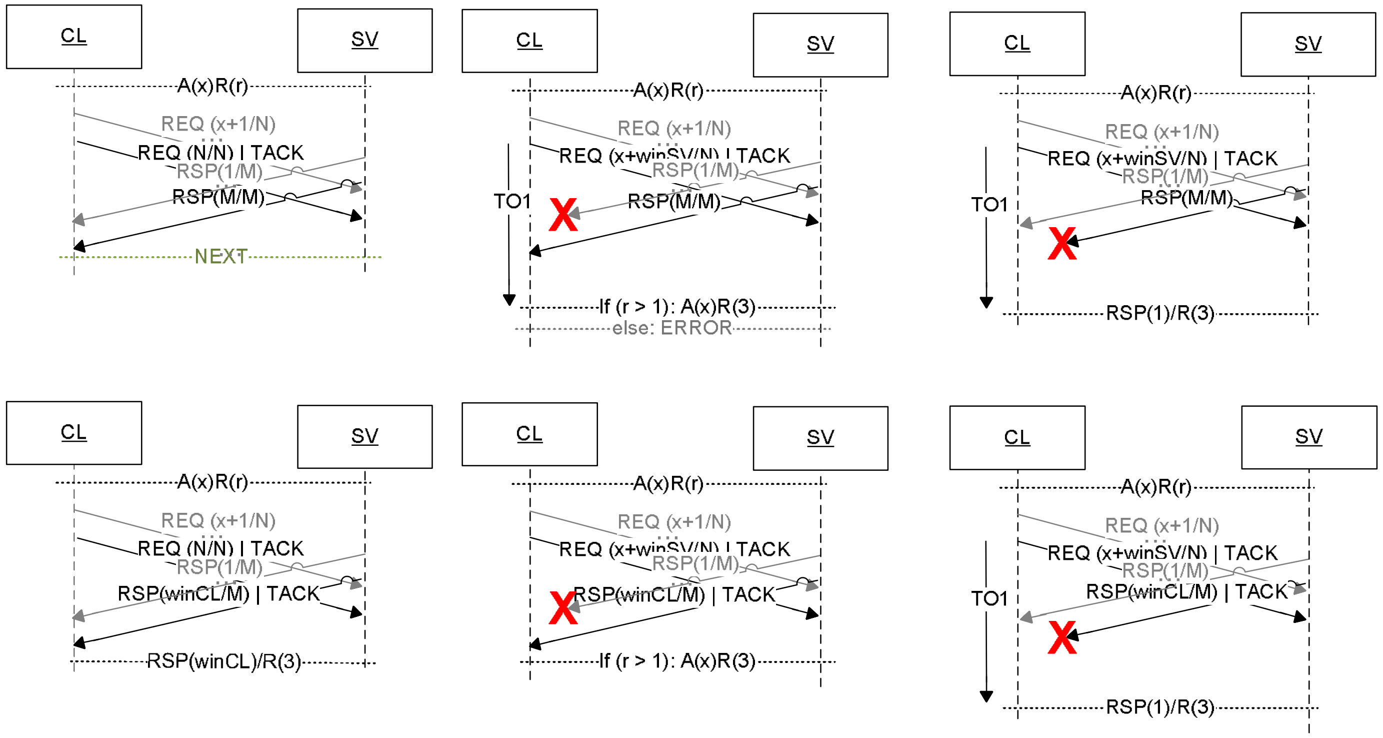
A(x)R(r)



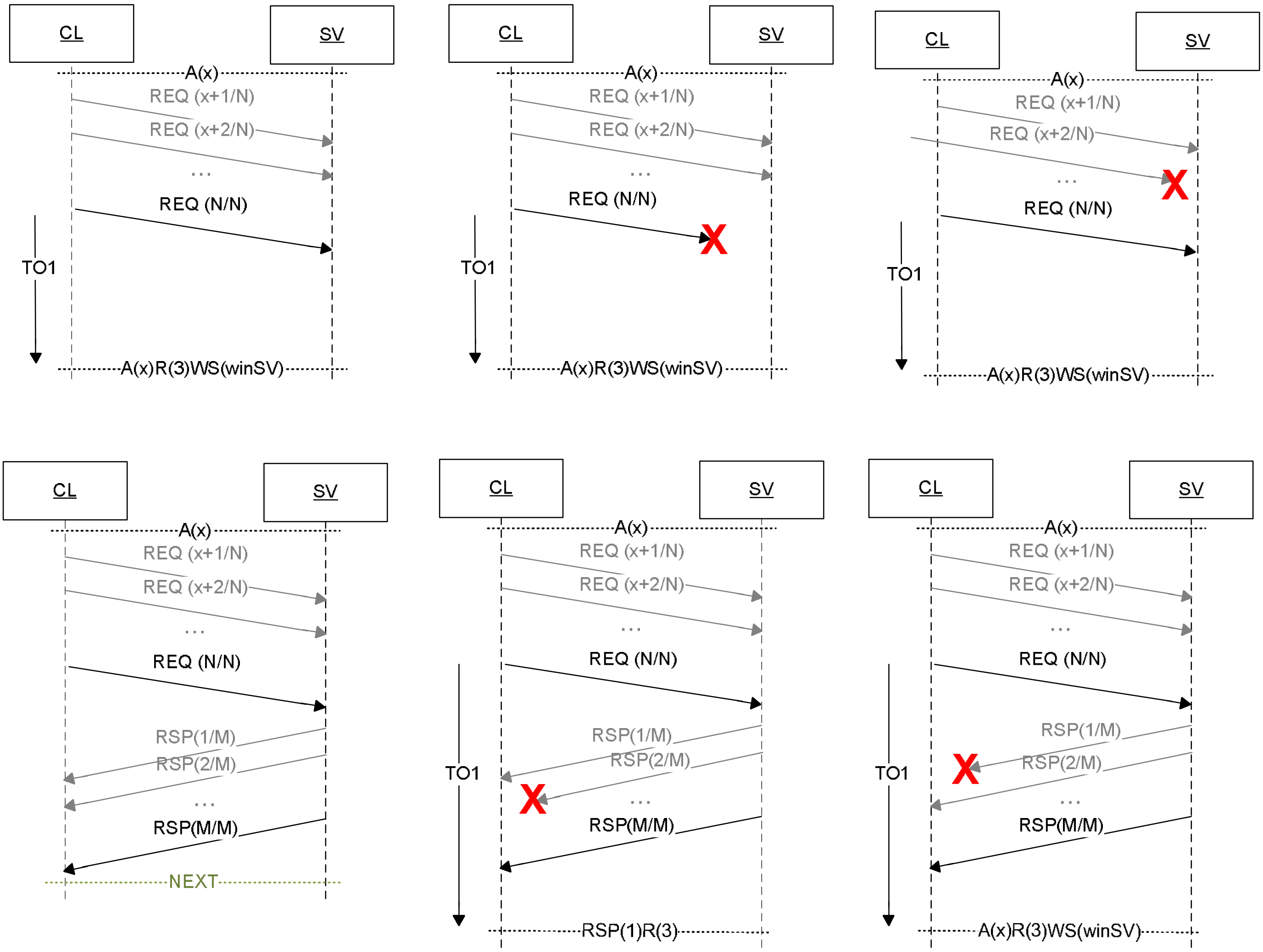
Without Intersection

x+w >= N

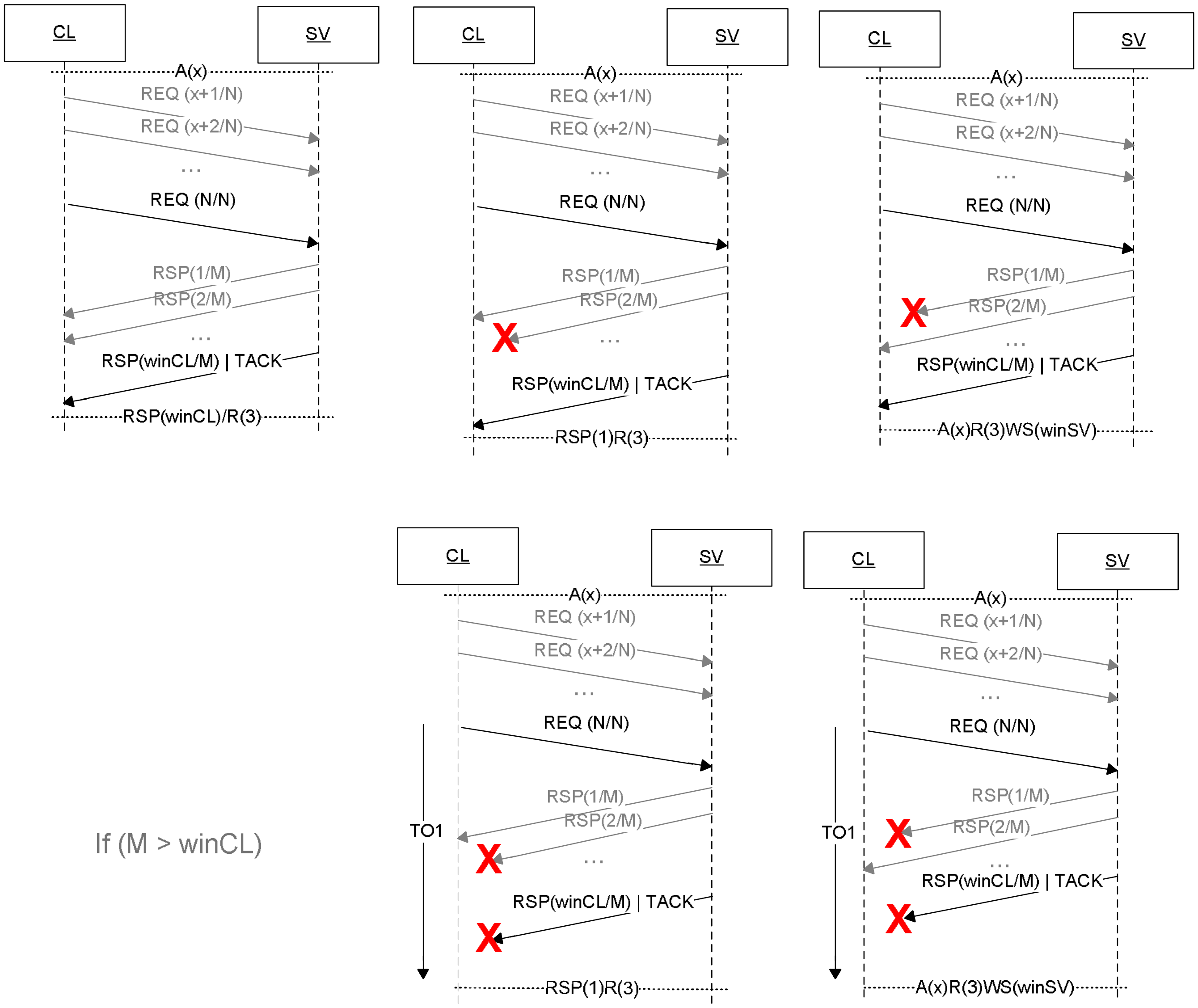
With Intersection

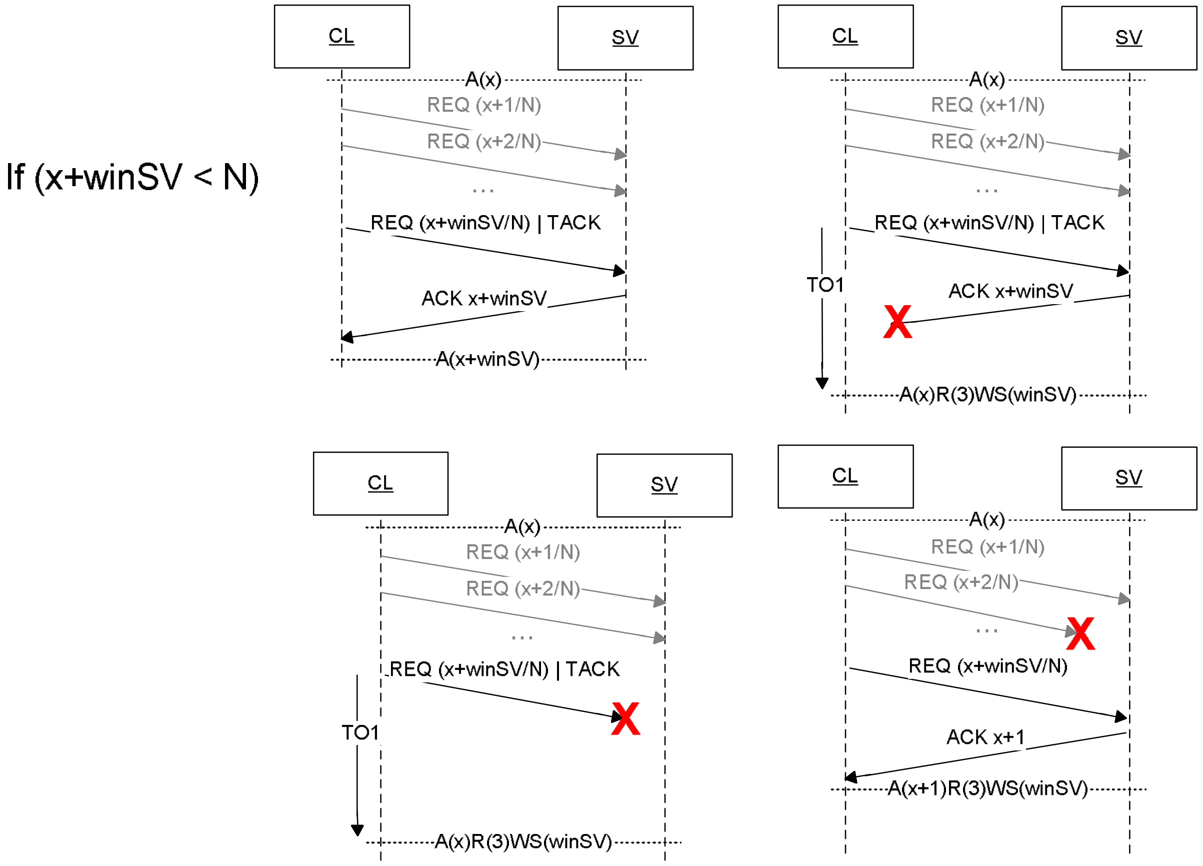


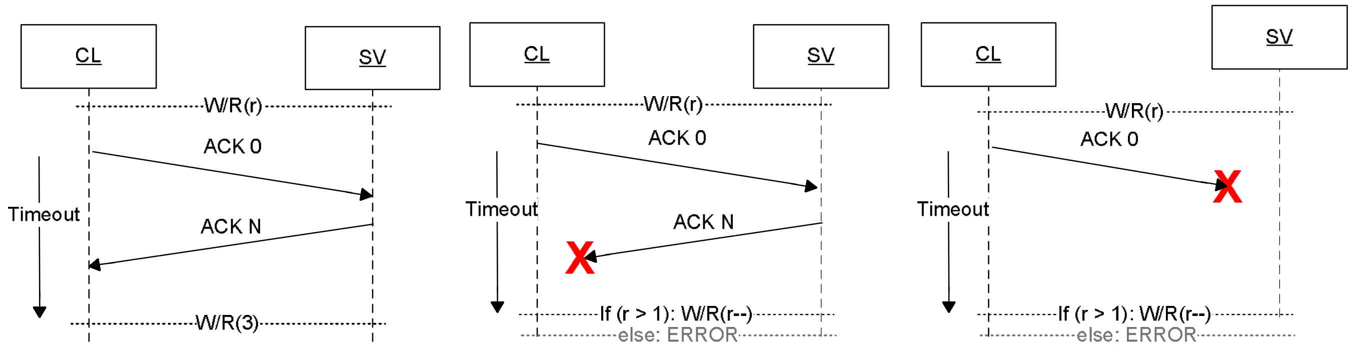
If (x+winSV >= N) A(X)



If (M <= winCL)


CL
SV
A(x)
REQ (x+1/N)
REQ (x+2/N)
…
REQ (N/N)
RSP(1/M)
RSP(2/M)
…
RSP(winCL/M) | TACK
RSP(winCL)/R(3)
CL
SV
A(x)
REQ (x+1/N)
REQ (x+2/N)
…
REQ (N/N)
RSP(1/M)
RSP(2/M)
X
…
RSP(winCL/M) | TACK
RSP(1)R(3)
CL
SV
A(x)
REQ (x+1/N)
REQ (x+2/N)
…
REQ (N/N)
RSP(1/M)
X
RSP(2/M)
…
RSP(winCL/M) | TACK
A(x)R(3)WS(winSV)
If (M > winCL)
CL
SV
A(x)
REQ (x+1/N)
REQ (x+2/N)
…
REQ (N/N)
TO1
RSP(1/M)
RSP(2/M)
X
…
RSP(winCL/M) | TACK
X
RSP(1)R(3)
CL
SV
A(x)
REQ (x+1/N)
REQ (x+2/N)
…
REQ (N/N)
TO1
RSP(1/M)
X
RSP(2/M)
…
RSP(winCL/M) | TACK
X
A(x)R(3)WS(winSV)

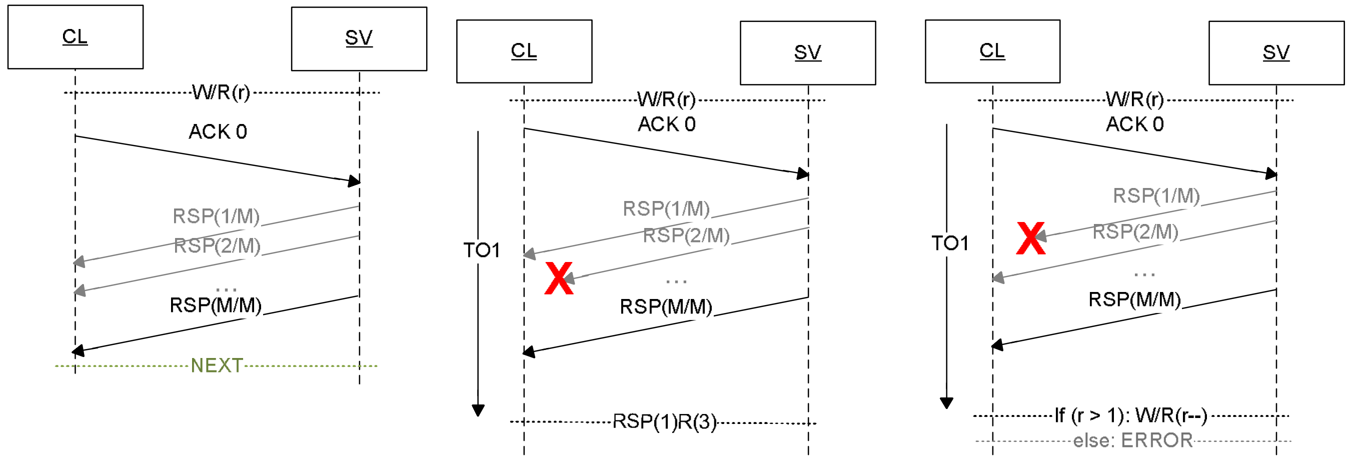

If (x+winSV < N)
CL
SV
A(x)
REQ (x+1/N)
REQ (x+2/N)
…
REQ (x+winSV/N) | TACK
ACK x+winSV
A(x+winSV)
CL
SV
A(x)
REQ (x+1/N)
REQ (x+2/N)
…
REQ (x+winSV/N) | TACK
TO1
ACK x+winSV
X
A(x)R(3)WS(winSV)
CL
SV
A(x)
REQ (x+1/N)
REQ (x+2/N)
…
REQ (x+winSV/N) | TACK
TO1
X
A(x)R(3)WS(winSV)
CL
SV
A(x)
REQ (x+1/N)
REQ (x+2/N)
…
X
REQ (x+winSV/N)
ACK x+1
A(x+1)R(3)WS(winSV)



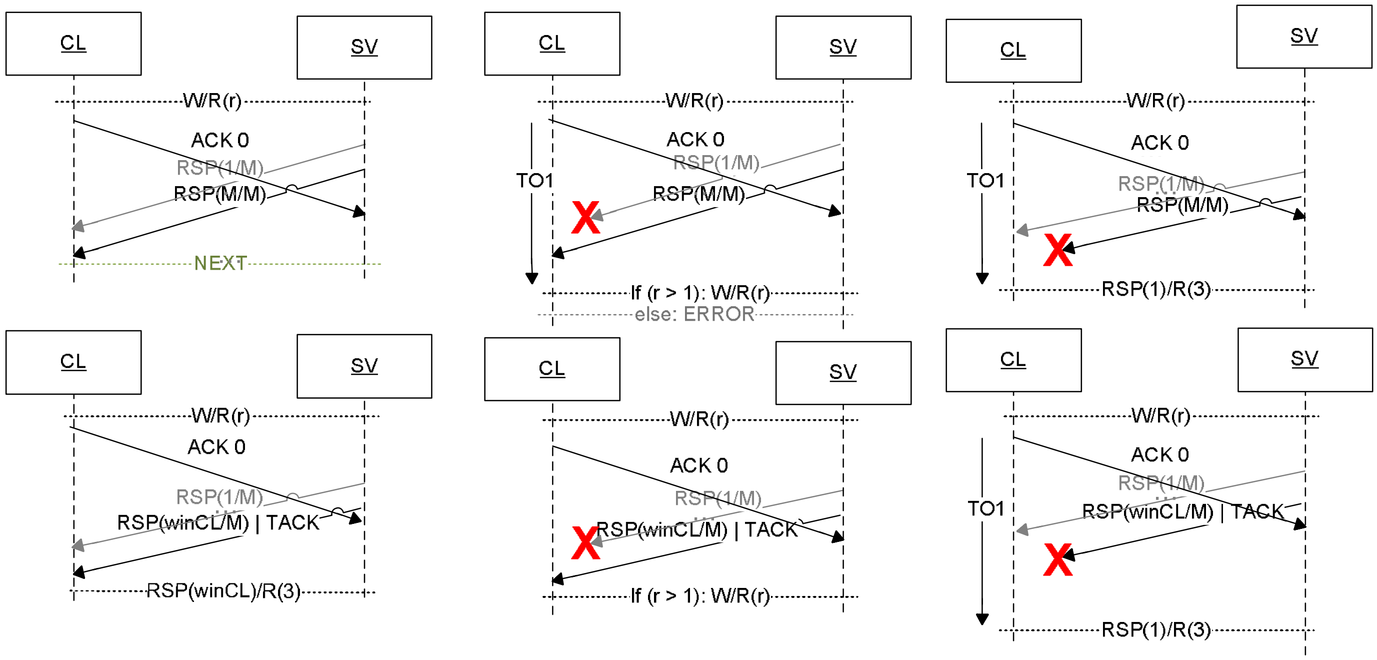
W/R(r): Working and r retries with ping until ERROR

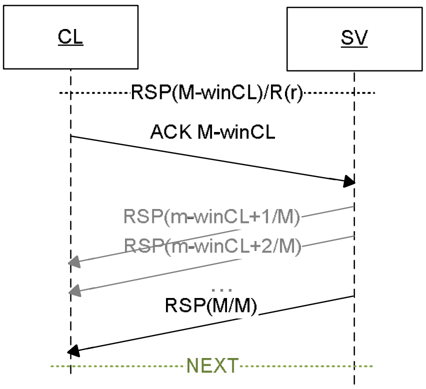
No Response

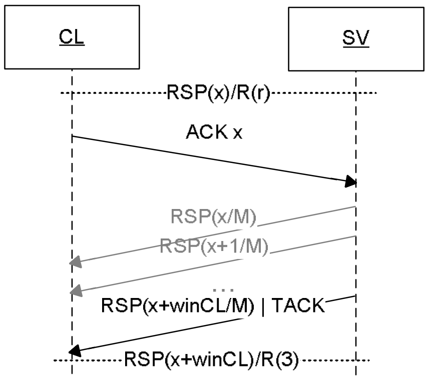
Response without intersection



Response with intersection



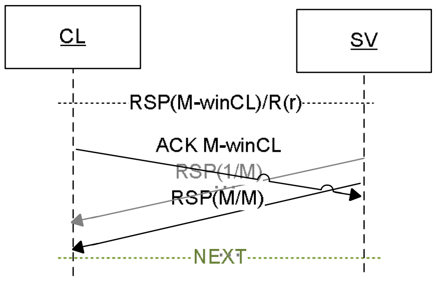


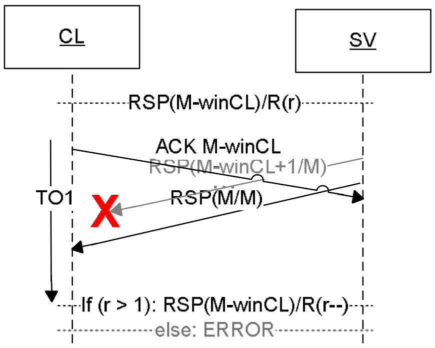


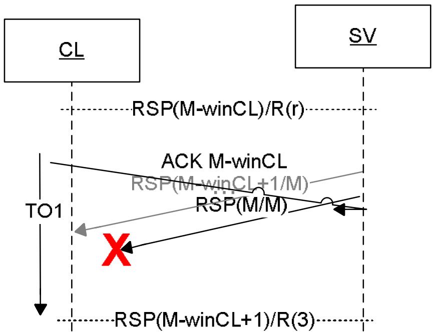
RSP(x)/R(r)
x fragments of
Response
transported. r
retries

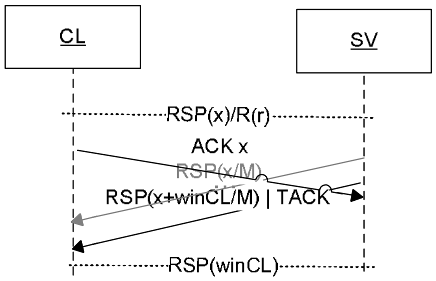
Without intersection

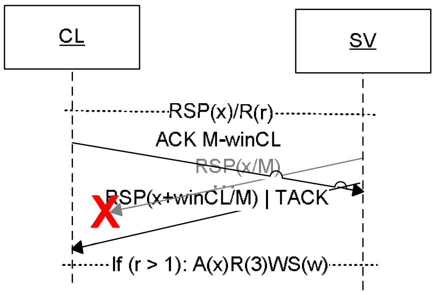
With intersection

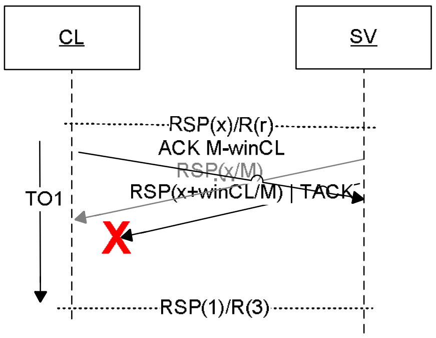


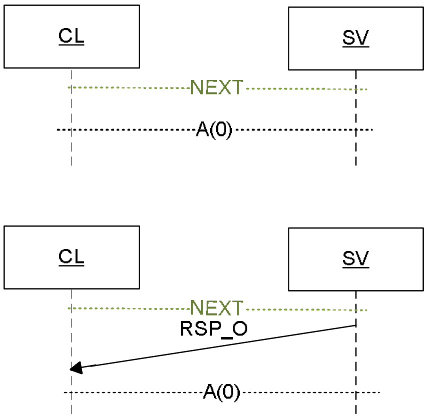










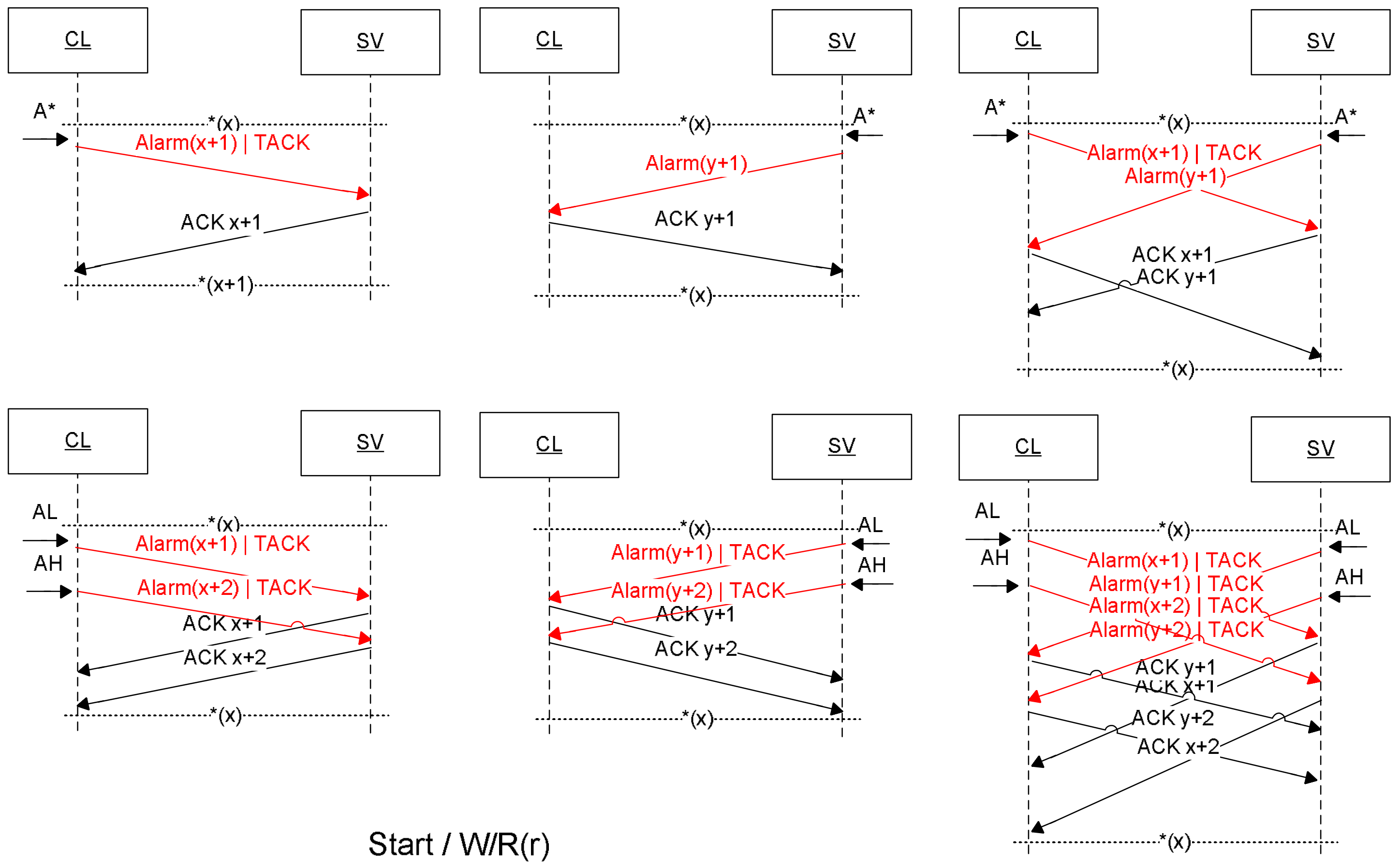


RSP_O == RSP(x/M)

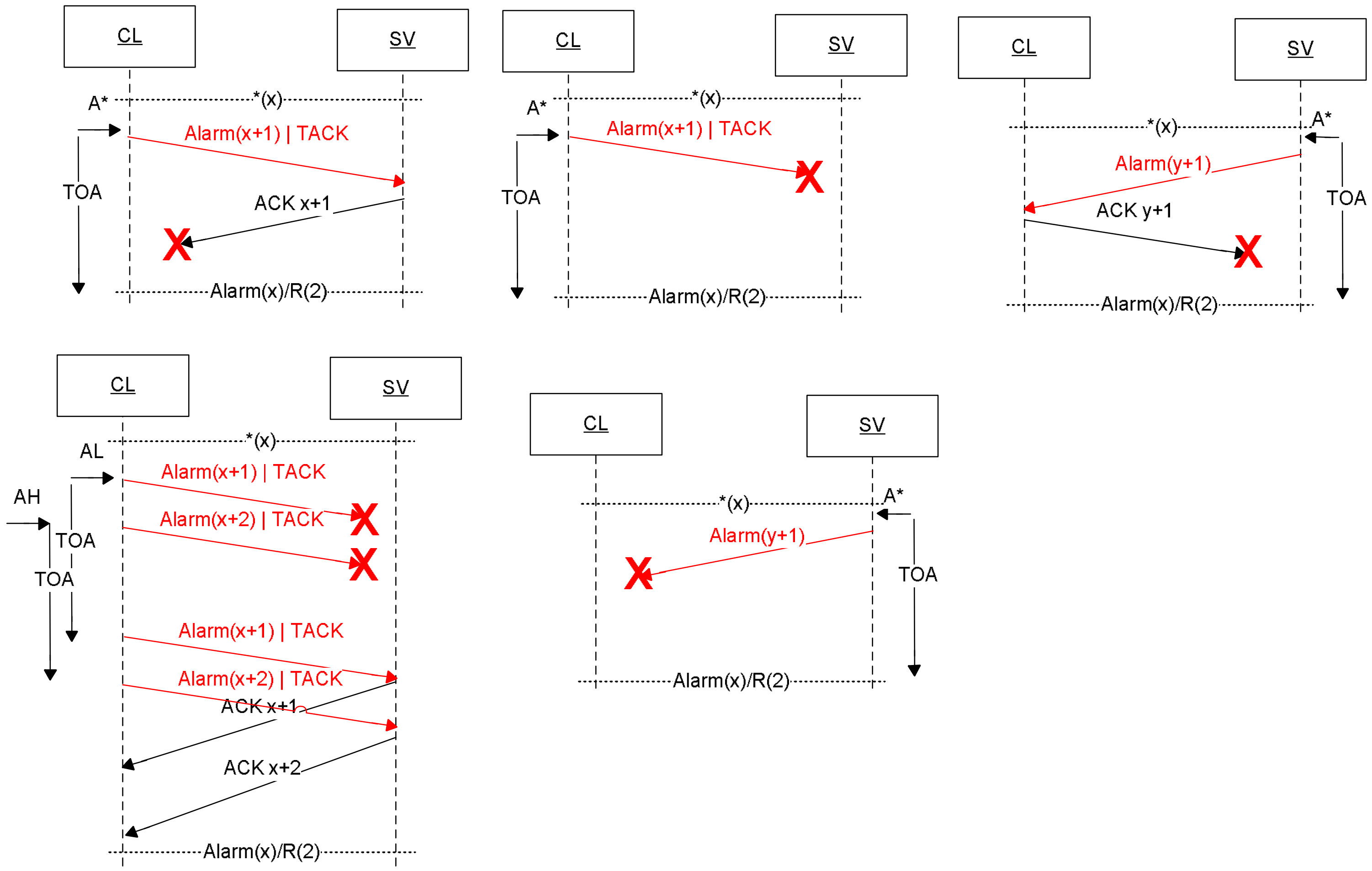
NEXT conforms A(0) when CL starts a call.
Old responses (RSP_O) are not observed.

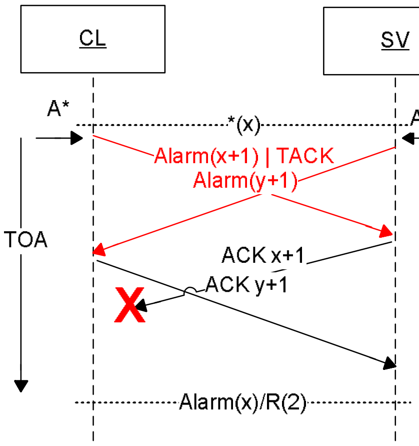
Appendix B.

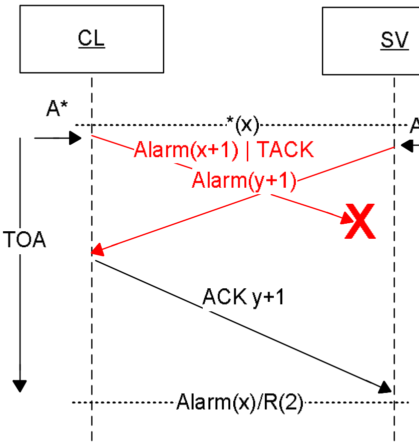
RTA Fragmentation Variant Advanced - Sequence

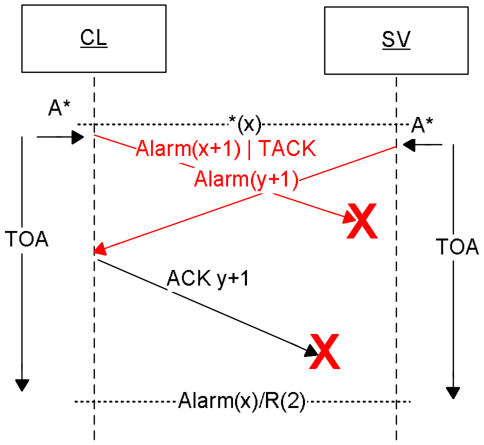

CL
SV
A*
*(x)
Alarm(x+1) | TACK
ACK x+1
*(x+1)
CL
SV
*(x)
A*
Alarm(y+1)
ACK y+1
*(x)
CL
SV
A*
*(x)
A*
Alarm(x+1) | TACK
Alarm(y+1)
ACK x+1
ACK y+1
*(x)
CL
SV
AL
*(x)
Alarm(x+1) | TACK
AH
Alarm(x+2) | TACK
ACK x+1
ACK x+2
*(x)
CL
SV
*(x)
AL
Alarm(y+1) | TACK
AH
Alarm(y+2) | TACK
ACK y+1
ACK y+2
*(x)
CL
SV
AL
*(x)
AL
AH
Alarm(x+1) | TACK
Alarm(y+1) | TACK
Alarm(x+2) | TACK
AH
Alarm(y+2) | TACK
ACK y+1
ACK x+1
ACK y+2
ACK x+2
*(x)

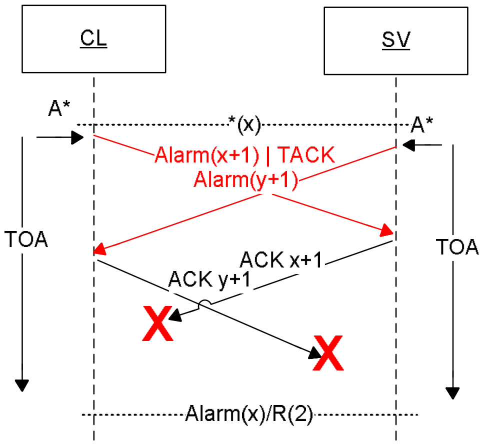
Start / W/R(r)

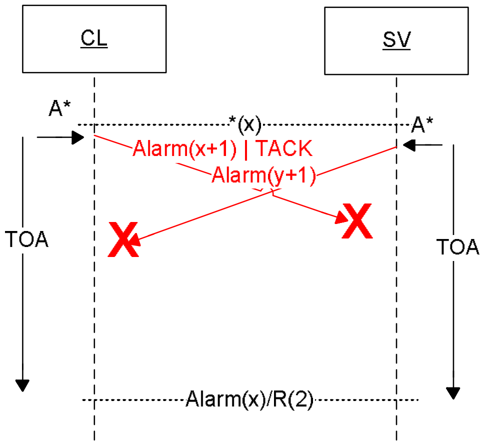

CL
SV
A*
*(x)
Alarm(x+1) | TACK
TOA
ACK x+1
X
Alarm(x)/R(2)
CL
SV
A*
*(x)
Alarm(x+1) | TACK
TOA
X
Alarm(x)/R(2)
CL
SV
*(x)
A*
Alarm(y+1)
TOA
ACK y+1
X
Alarm(x)/R(2)
CL
SV
*(x)
AL
AH
Alarm(x+1) | TACK
Alarm(x+2) | TACK
TOA
TOA
X
X
Alarm(x+1) | TACK
Alarm(x+2) | TACK
ACK x+1
ACK x+2
Alarm(x)/R(2)
CL
SV
*(x)
A*
Alarm(y+1)
X
TOA
Alarm(x)/R(2)



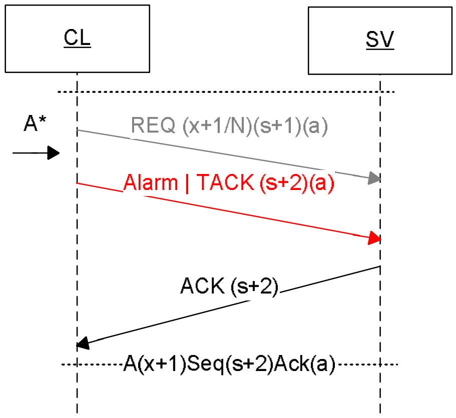


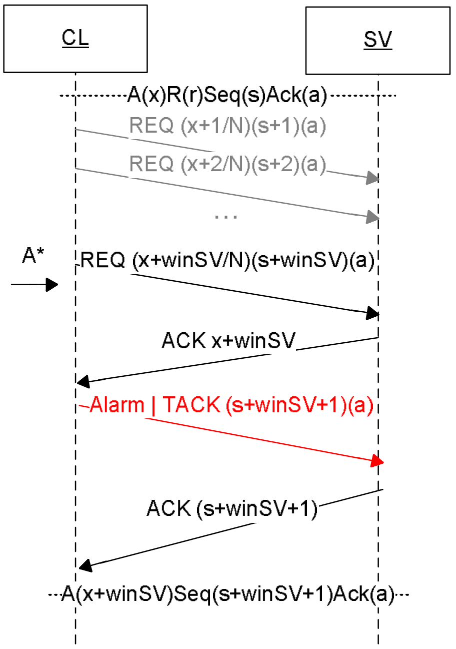


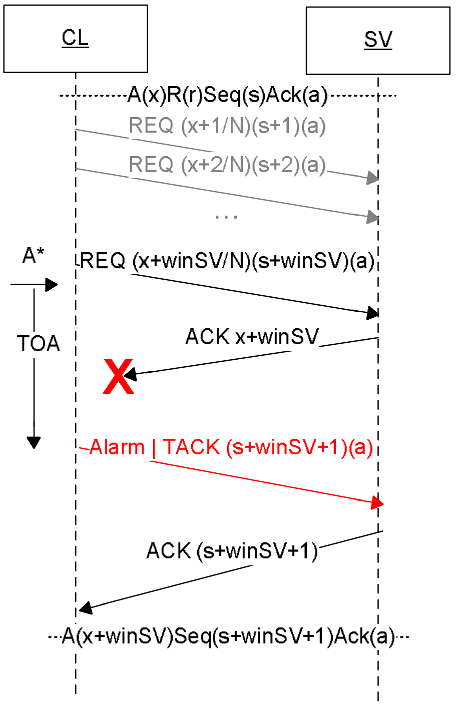


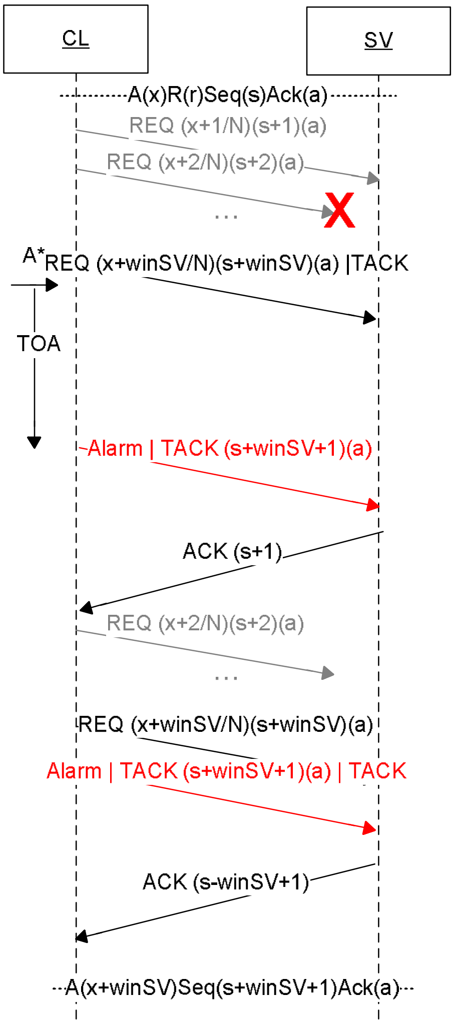


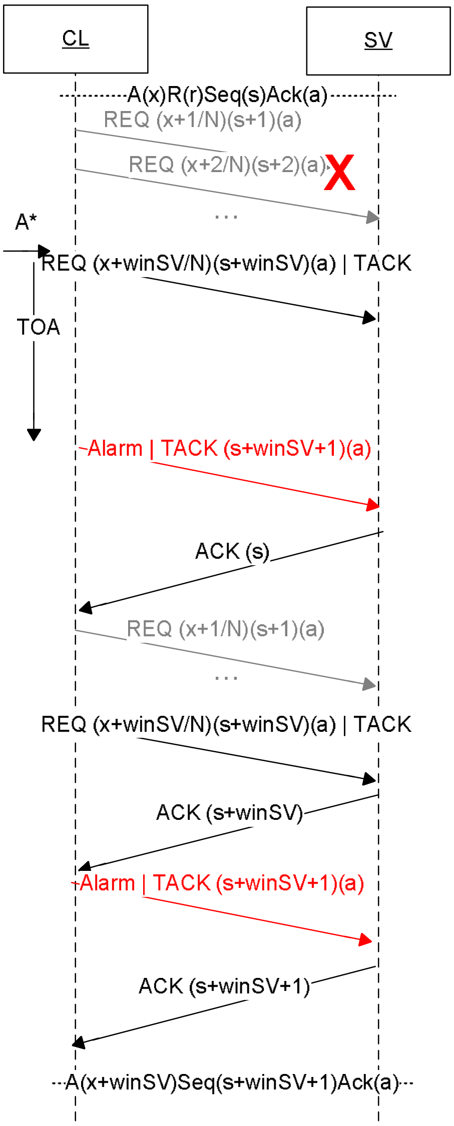
Start, A(x)R(R), A(x), NEXT



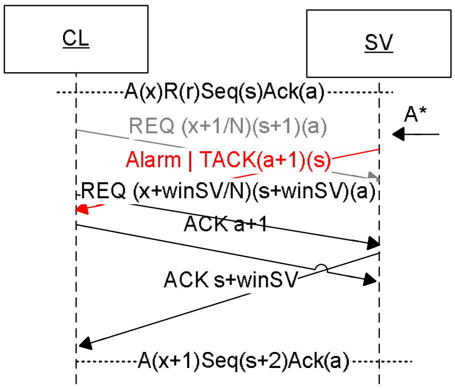


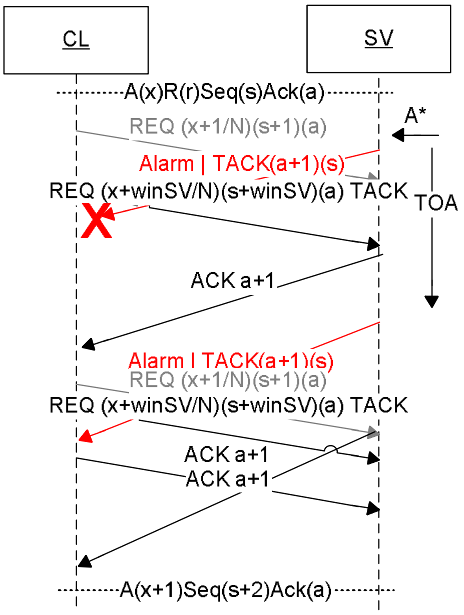


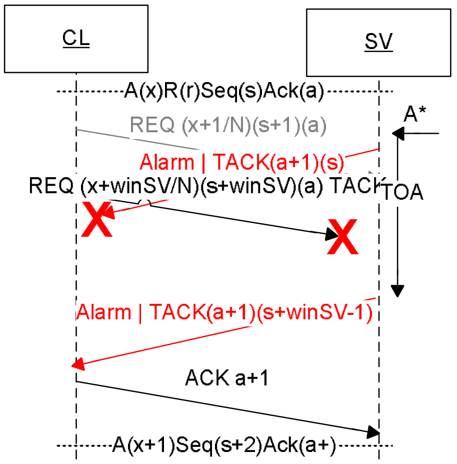

CL
SV
A(x)R(r)Seq(s)Ack(a)
REQ (x+1/N)(s+1)(a)
REQ (x+2/N)(s+2)(a)
...
A*
REQ (x+winSV/N)(s+winSV)(a) |TACK
TOA
Alarm | TACK (s+winSV+1)(a)
ACK (s+1)
REQ (x+2/N)(s+2)(a)
...
REQ (x+winSV/N)(s+winSV)(a)
Alarm | TACK (s+winSV+1)(a) | TACK
ACK (s-winSV+1)
A(x+winSV)Seq(s+winSV+1)Ack(a)

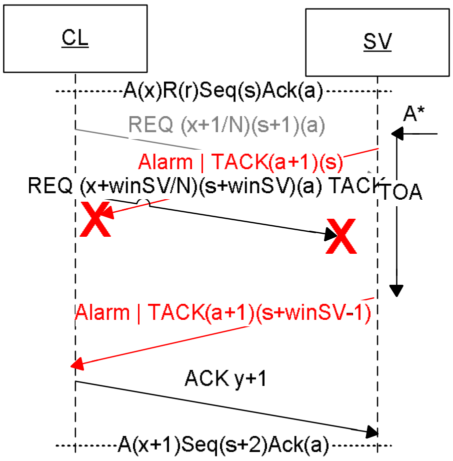

CL
SV
A(x)R(r)Seq(s)Ack(a)
REQ (x+1/N)(s+1)(a)
REQ (x+2/N)(s+2)(a)
...
A*
REQ (x+winSV/N)(s+winSV)(a) | TACK
TOA
Alarm | TACK (s+winSV+1)(a)
ACK (s)
REQ (x+1/N)(s+1)(a)
...
REQ (x+winSV/N)(s+winSV)(a) | TACK
ACK (s+winSV)
Alarm | TACK (s+winSV+1)(a)
ACK (s+winSV+1)
A(x+winSV)Seq(s+winSV+1)Ack(a)

Start, A(x)R(R), A(x), NEXT









Appendix C.

Initiator State Machine of the Basic Variant.

The initiator state machines have been written by Thomas Pössler. He has used as basis an early state machine by me as the base for this work. (Pössler and Stefan, 2017a).

#	Current State	Event	Condition	Actions taken	Output	Next State
1	UNBOUND	RSI_Call.req	OpNum in [Connect, ReadImplicit, ReadConnectionless]	/* init new session */ isap.rsap := CON-SAP SendSeqNum := 0xFFFE SendWindowSize := 2 /* implementation specific */ RecvAckNum := 0xFFFE RecvSendNum := 0xFFFE RecvWindowSize := 1 SendCallSequence := 0x07 /* init call */ TimeoutCount := 0 RecvFragOffset := 0 SendCallSequence += 1 M_PrepareRequest (OpNum, ArgsRspMaxLength, ArgsReqLenght, ArgsReq) StartTimer (ApplicationReadyTimeout) KeepSendSeqNum := SendSeqNum SendSeqNum += 1	M_SendFreq_req	FREQ
2	UNBOUND	RSI_Call.req	OpNum not-in [Connect, ReadImplicit, ReadConnectionless]		RSI_Call.cnf (-)	UNBOUND
3	UNBOUND	RSI_Abort.req			RSI_Abort.cnf (-)	UNBOUND
4	UNBOUND	RSI_Ack_ind		ignore		UNBOUND
5	UNBOUND	RSI_Fres_ind		ignore		UNBOUND
6	UNBOUND	RSI_Error_ind		ignore		UNBOUND
7	UNBOUND	TimerExpired (ANY)		ignore		UNBOUND
8	UNBOUND	M_SendAck_cnf		ignore		UNBOUND
9	UNBOUND	M_SendFreq_cnf		ignore		UNBOUND
10	UNBOUND	M_SendError_cnf		ignore		UNBOUND
11	IDLE	RSI_Call.req	OpNum in [Read, Write, Control, ReadNotification, PrmWrite]	TimeoutCount := 0 RecvFragOffset := 0 SendCallSequence += 1 M_PrepareRequest (OpNum, ArgsRspMaxLength, ArgsReqLenght, ArgsReq) StartTimer (ApplicationReadyTimeout) KeepSendSeqNum := SendSeqNum SendSeqNum += 1	M_SendFreq_req	FREQ
12	IDLE	RSI_Call.req	ReadImplicit, ReadConnectionless]		RSI_Call.cnf (-)	IDLE
13	IDLE	RSI_Abort.req			M_SendError_req	UNBOUND
14	IDLE	RSI_Ack_ind		ignore		IDLE
15	IDLE	RSI_Fres_ind		ignore		IDLE
16	IDLE	RSI_Error_ind			RSI_Abort.ind	IDLE
17	IDLE	TimerExpired (ANY)		ignore		IDLE
18	IDLE	M_SendAck_cnf		ignore		IDLE
19	IDLE	M_SendFreq_cnf		ignore		IDLE
20	IDLE	M_SendError_cnf		ignore		IDLE
21	FREQ	Do_CheckMoreToSend	M_CheckMoreToSend == LastFragAcked	StopTimer (FragTimeout) /* for send */ StartTimer (FragTimeout) /* for receive */ TimeoutCount := 0		DRSP
22	FREQ	Do_CheckMoreToSend	M_CheckMoreToSend ==	SendSeqNum += 1	M_SendFreq_req	FREQ
23	FREQ	Do_CheckMoreToSend	M_CheckMoreToSend == WaitAck	/* Note: Send a frag with TACK set or send the last frag, wait for Ack */		FREQ

24	FREQ	Do_CheckMoreToSend	M_CheckMoreToSend == Timeout	/* Note: more than MaxFragTimeoutCount Timeouts occurs without response */	RSI_Call.cnf(-)	IDLE
25	FREQ	M_SendFreq_cnf		StartTimer (FragTimeout)	Do_CheckMoreToSend (SendCnf)	FREQ
26	FREQ	TimerExpired (FragTimeout)		TimeoutCount += 1	Do_CheckMoreToSend (Timeout)	FREQ
27	FREQ	M_SendAck_cnf		StartTimer (FragTimeout)		FREQ
28	FREQ	RSI_Ack_ind		M_UpdateAckInfo TimeoutCount := 0	if TACK /* because of implicit Ack */ M_SendAck_req Do_CheckMoreToSend	FREQ
29	FREQ	RSI_Call.req			RSI_Call.cnf(-)	FREQ
30	FREQ	RSI_Abort.req			RSI_Call.cnf(-) M_SendError_req	UNBOUND
31	FREQ	RSI_Fres_ind		ignore		FREQ
32	FREQ	RSI_Error_ind		StopTimer (ANY)	RSI_Abort.ind RSI_Call.cnf(-)	IDLE
33	FREQ	M_SendError_cnf		ignore		FREQ
34	DRSP	Do_LastFrag_ind	OpNum in [ReadImplicit, ReadConnectionless]	StopTimer (ANY) PNIOStatus := KeepPNIOStatus ArgsRspLength := RecvFragOffset - PDUOffset	RSI_Call.cnf(+)	UNBOUND
35	DRSP	Do_LastFrag_ind	OpNum not-in [ReadImplicit, ReadConnectionless]	StopTimer (ANY) PNIOStatus := KeepPNIOStatus ArgsRspLength := RecvFragOffset - PDUOffset	RSI_Call.cnf(+)	IDLE
36	DRSP	RSI_Fres_ind	PDU.SendSeqNum == (RecvSeqNum + 1) && PDU.FragOffset == RecvFragOffset && M_MoreFresCheck	/* Expected, do defragmentation */ ArgsRsp := M_Copy_FragData RecvSeqNum := PDU.SendSeqNum RecvFragOffset += PDU.VarParLen - 4	if isLastFrag Do_LastFrag_ind	DRSP
37	DRSP	RSI_Fres_ind	! (PDU.SendSeqNum == (RecvSeqNum + 1) && PDU.FragOffset == RecvFragOffset	/* ! Expected, Ignore */ ignore		DRSP
38	DRSP	TimerExpired	! (TimeoutCount >	TimeoutCount += 1	M_SendAck_req	DRSP
39	DRSP	TimerExpired	TimeoutCount >	StopTimer (ANY)	RSI_Call.cnf(-)	IDLE
40	DRSP	TimerExpired (ApplicationReadyTimeout)		StopTimer (ANY)	RSI_Call.cnf(-)	IDLE
41	DRSP	RSI_Ack_ind		TimeoutCount := 0 M_UpdateAckInfo	if TACK M_SendAck_req	DRSP
42	DRSP	M_SendAck_cnf		StartTimer (FragTimeout)		DRSP
43	DRSP	RSI_Error_ind		StopTimer (ANY)	RSI_Abort.ind RSI_Call.cnf(-)	IDLE
44	DRSP	RSI_Call.req			RSI_Call.cnf(-)	DRSP
45	DRSP	RSI_Abort.req		StopTimer (ANY)	RSI_Call.cnf(-) M_SendError_req	UNBOUND
46	DRSP	M_SendFreq_cnf		ignore		DRSP
47	DRSP	M_SendError_cnf		ignore		DRSP

Name	Type	Meaning
MaxFragTimeoutCount	Const	This value determines how often a TimerExpired (FragTimeout) may occur. Value: 3. Note: see [Protocol::Timeout Frag]
MaxVarPartLen	Const	The constant value 1432, see RTA-PDU::VarPartLen
FragTimeout	Timer	Timeout for retransmission, Timeout for busy indicator. Value: 2 seconds. Note: see [Protocol::Timeout Frag]
RemoteApplicationTimeout	Timer	Timeout for finishing a RSI-Call. Value: 300 seconds. Note: see [Profile::Remote_Application_Timeout]
TimeoutCount	Value	Counts the TimerExpired events.
StartTimer	Function	This function is used to start or restart a timer.
StopTimer	Function	This function is used to stop a timer.
TimerExpired	Function	This function signals that a timer is expired.
isap	Value	This RSI-Initiator, refer to RSI-LMPM::Rsi-List
SendSeqNum	Value	The largest SendSeqNum sent
SendWindowSize	Value	The WindowSize of this RSI-Initiator
RecvAckNum	Value	The largest AckSeqNum received
RecvSendNum	Value	The largest SendSeqNum reveiced
RecvFragOffset	Value	The next FragOffset to receive, points to CallBuffer
RecvWindowSize	Value	The tailored WindowSize of the RSI-Responder /* The WindowSize within the first ACK or Connect response fragment will be taken,
SendCallSequence	Value	CallSequence; Range (0..7); += 1 does an wrap around
KeepSendSeqNum	Value	Keep the SendSeqNum for re-transmitting the first REQ Fragment
KeepPNIOStatus	Value	Keep PNIOStatus from first RES Fragment
PDUOffset	Macro	// The fragmented part of the "PROFINETIOServiceResPDU" starts at this Offset within the RSI-PDU
SendArray	Value	Array of 1 to n request fragment to send SendArray.Length is the count of fragments fragments ::= (sendSeqNum, sendTack, sendMoreFrag, offset, dataLen, Data[])

SearchIndexWithSendSeqNum	Function	// Return the index within SendArray with SendArray[idx].SendSeqNum == SendSeqNum
		for (idx := 0; idx < SendArray.Length; ++idx)
		   if (SendArray[idx].SendSeqNum == SendSeqNum) return idx
		endfor
		return undef // shall never happen

SearchLastAckedSendSeqNum	Function	// Return the highest acknowledged SendSeqNum within SendArray. Return KeepSendSeqNum if no entry is acknowledged.
		for (high_idx := 0, idx := 0; idx < SendArray.Length; ++idx)
		   if (SendArray[idx].SendSeqNum == RecvAckSeqNum) return RecvAckSeqNum
		endfor
		return KeepSendSeqNum

build	Macro	Marshal a RSI-REQ-PDU from it's arguments

M_PrepareRequest	Macro	// Prepare the args from RSI_Call.req to an array of FREQ-RTA-PDU to send
		RSI-REQ-PDU Data := build (OpNum, ArgsRspMaxLength, ArgsReqLength, ArgsReq)
		DataLen := sizeOf (Data)
		FragPartLen := MaxVarPartLen - 4/*sizeOf FOpnumOffset*/
		
		SendArray := []
		ssn := SendSeqNum
		offset := 0
		for (len := 0, i := 0; len < DataLen; len += FragPartLen, i += 1)
		  ssn += 1
		  if (len + FragPartLen < DataLen)
		    sendTack := ((i+1) % RecvWindowSize) == 0 ? 1 : 0
		    SendArray[i] := (ssn, sendTack, sendMoreFrag := 1, offset,
		                FragPartLen, Data[len .. len + FragPartLen - 1])
		    offset += FragPartLen
		  else
		    rest := DataLen - len
		    SendArray[i] := (ssn, sendTack := 0, sendMoreFrag := 0, offset
		                rest, Data[len .. len + rest - 1])
		  endif
		endfor

Do_LastFrag_ind	Event	Signal an internal Event
Do_CheckMoreToSend(Trigger)	Event	Signal an internal Event with Trigger as Parameter. This Parameter is used by

M_CheckMoreToSend()	Macro	// SendSeqNum was send; check if there is more to send; returns one of:
		//- Send: send next FRAG of prepared array
		//- WaitAck: wait for ACK, has more FRAGs to send or wait for ACK of the last FRAG
		//- LastFragAcked: the last FRAG from array was acked
		//- Timeout: timeout too often, cancel call
		idx := SendArray.SearchIndexWithSendSeqNum (SendSeqNum)
		
		if (Trigger == SendCnf)
		  if (SendArray[idx].sendMoreFrag == 0) return WaitAck
		  if (SendArray[idx].sendTack == 1) return WaitAck
		  return Send
		
		if (Trigger == Timeout)
		  if (TimeoutCount > MaxFragTimeoutCount) return Timeout
		  SendSeqNum := SendArray.SearchLastAckedSendSeqNum ()
		  return Send
		
		 if (Trigger == Ack)
		  if (SendArray[idx].sendMoreFrag == 0)
		    if (SendArray[idx].SendSeqNum == RecvAckNum) return LastFragAcked
		    return WaitAck
		  elseif (SendArray[idx].sendTack == 1)
		    if (SendArray[idx].SendSeqNum == RecvAckNum) return Send
		    return WaitAck
		  endif
		  SendSeqNum := SendArray.SearchLastAckedSendSeqNum ()
		  return Send

Appendix D.

Responder State Machine of the Basic Variant.

The responder state machines have been written by Thomas Pössler. He has used as basis an early state machine by me as the base for this work. (Pössler and Stefan, 2017b).

#	Current State	Event	Condition	Actions taken	Output	Next State
1	IDLE	Do_Connect_ind		SendSeqNum := 0xFFFE SendWindowSize := 2 /* implementation specific */ RecvAckNum := 0xFFFE RecvSendNum := PDU.SendSeqNum - 1 RecvFragOffset := 0 RecvWindowSize := 0 ConnectResponseState := IDLE KeepConnectSendSeqNum := PDU.SendSeqNum - 1 M_do_fullyBound StartTimer (FragTimeout)		DREQ
2	IDLE	RSI_Connect_ind			Do_Connect_ind	IDLE
3	IDLE	RSI_Freq_ind		ignore		IDLE
4	IDLE	RSI_Ack_ind		ignore		IDLE
5	IDLE	RSI_Call.rsp ()		ignore		IDLE
6	IDLE	M_SendFres_cnf		ignore		IDLE
7	IDLE	M_SendError_cnf		ignore		IDLE
8	IDLE	M_SendAck_cnf		ignore		IDLE
9	IDLE	TimerExpired		ignore		IDLE
10	IDLE	RSI_Error_ind		ignore		IDLE
11	IDLE	RSI_Abort.req ()			RSI_Abort.cnf (-)	IDLE
12	DREQ	Do_LastFrag_ind		StopTimer (FragTimeout) ArgsReqLength := RecvFragOffset - PDUOffset OpNum := KeepOpNum ArgsRspMax := KeepArgsRspMax if (ConnectResponseState == IDLE) if (isFirstFrag && isLastFrag) /* single frag connect */ ConnectResponseState := PING else ConnectResponseState := ABORT	RSI_Call.ind ()	WRSP
13	DREQ	RSI_Connect_ind	M_CheckConnect == ABORT		RSI_Abort.ind ()	ABORTING
14	DREQ	RSI_Connect_ind	M_CheckConnect != ABORT		/* Setup a new RSI_Connect_ind, RSI_Ack_ind and RSI_Frag_ind comes	IDLE
15	DREQ	RSI_Freq_ind	PDU.SendSeqNum == (RecvSeqNum + 1) && PDU.FragOffset == RecvFragOffset && M_MoreFreqCheck	/* Expected: do defragmentation */ ArgsReq := M_Copy_FragData RecvSeqNum := PDU.SendSeqNum RecvFragOffset += PDU.VarParLen - 4 StartTimer (FragTimeout)	if (isLastFrag) Do_LastFrag_ind	DREQ
16	DREQ	RSI_Freq_ind	! (PDU.SendSeqNum == (RecvSeqNum + 1) && PDU.FragOffset == RecvFragOffset	/* ! Expected: Ignore */ ignore		DREQ
17	DREQ	RSI_Ack_ind		M_UpdateAckInfo	if (TACK) M_SendAck_req	DREQ
18	DREQ	RSI_Call.rsp ()		ignore		DREQ
19	DREQ	M_SendFres_cnf		ignore		DREQ
20	DREQ	M_SendError_cnf		ignore		DREQ
21	DREQ	M_SendAck_cnf		ignore		DREQ
22	DREQ	TimerExpired (FragTimeout)	ConnectResponseState != IDLE		RSI_Abort.ind ()	ABORTING
23	DREQ	TimerExpired (FragTimeout)	ConnectResponseState == IDLE	M_do_halfBound		IDLE
24	DREQ	RSI_Error_ind	ConnectResponseState != IDLE		RSI_Abort.ind ()	ABORTING

25	DREQ	RSI_Error_ind	ConnectResponseState == IDLE	M_do_halfBound StopTimer (FragTimeout)		IDLE
26	DREQ	RSI_Abort.req ()			M_SendError_req	ABORTING
27	WRSP	RSI_Call.rsp ()		if (SendSeqNum == 0xFFFE) SendSeqNum := KeepConnectSendSeqNum M_PrepareResponse (PNIOStatus, ArgsRsp, ArgsRspLength) StartTimer (FragTimeout) SendSeqNum := SendSeqNum + 1	M_SendFres_req	FRSP
28	WRSP	RSI_Connect_ind	M_CheckConnect == ABORT		RSI_Abort.ind ()	ABORTING
29	WRSP	RSI_Connect_ind	M_CheckConnect != ABORT	ignore		WRSP
30	WRSP	RSI_Freq_ind		ignore		WRSP
31	WRSP	RSI_Ack_ind		M_UpdateAckInfo	if (TACK || (PDU.Type == ACK) || (PDU.Type == FRAG && PDU.SendSeqNum == RecvSeqNum)) M_SendAck_req	WRSP
32	WRSP	M_SendFres_cnf		ignore		WRSP
33	WRSP	M_SendError_cnf		ignore		WRSP
34	WRSP	M_SendAck_cnf		ignore		WRSP
35	WRSP	TimerExpired		ignore		WRSP
36	WRSP	RSI_Error_ind			RSI_Abort.ind ()	ABORTING
37	WRSP	RSI_Abort.req ()			M_SendError_req	ABORTING
38	FRSP	Do_CheckMoreToSend	M_CheckMoreToSend == LastFragAcked	RecvFragOffset := 0 /* Note: prepare next call */ if (ConnectResponseState == PING) ConnectResponseState := ABORT		DREQ
39	FRSP	Do_CheckMoreToSend	M_CheckMoreToSend == LastRIFragSend	/* ReadConnectionless or ReadImplicit */		ABORTING
40	FRSP	Do_CheckMoreToSend	M_CheckMoreToSend == Send	StartTimer (FragTimeout) SendSeqNum := SendSeqNum + 1	M_SendFres_req	FRSP
41	FRSP	Do_CheckMoreToSend	M_CheckMoreToSend == WaitAck	ignore		FRSP
42	FRSP	RSI_Connect_ind	M_CheckConnect == ABORT		RSI_Abort.ind ()	ABORTING
43	FRSP	RSI_Connect_ind	M_CheckConnect != ABORT	ignore		FRSP
44	FRSP	RSI_Freq_ind		ignore		FRSP
45	FRSP	RSI_Freq_ind		ignore		FRSP
46	FRSP	RSI_Ack_ind		M_UpdateAckInfo SendSeqNum := M_PrepareReSend	if (TACK) M_SendAck_req /* for implicit Ack_ind */ Do_CheckMoreToSend	FRSP
47	FRSP	RSI_Call.rsp ()		ignore		FRSP
48	FRSP	M_SendFres_cnf			Do_CheckMoreToSend	FRSP
49	FRSP	M_SendError_cnf		ignore		FRSP
50	FRSP	M_SendAck_cnf		ignore		FRSP
51	FRSP	TimerExpired (FragTimeout)	KeepOpNum in [ReadConnectionless,	/* Note: WaitAck */ M_do_halfBound		IDLE
52	FRSP	TimerExpired (FragTimeout)	KeepOpNum not-in [ReadConnectionless,	ignore		FRSP
53	FRSP	RSI_Error_ind			RSI_Abort.ind ()	ABORTING
54	FRSP	RSI_Abort.req ()			M_SendError_req	ABORTING
55	ABORTING	M_SendError_cnf		M_do_halfBound	RSI_Abort.cnf ()	IDLE
56	ABORTING	RSI_Connect_ind	KeepOpNum in [ReadConnectionless, ReadImplicit]		/* Setup a new RSI_Connect_ind, RSI_Ack_ind and RSI_Frag_ind comes	IDLE
57	ABORTING	RSI_Connect_ind	KeepOpNum not-in [ReadConnectionless,	ignore	/* must RSI_Abort.req first */	ABORTING
58	ABORTING	RSI_Freq_ind		ignore		ABORTING
59	ABORTING	RSI_Call.rsp ()		ignore		ABORTING

60	ABORTING	RSI_Ack_ind	KeepOpNum in [ReadConnectionless, ReadImplicit]	M_UpdateAckInfo SendSeqNum := M_PrepareReSend	if (TACK) M_SendAck_req /* for implicit Ack_ind */ Do_CheckMoreToSend	FRSP
61	ABORTING	RSI_Ack_ind	KeepOpNum not-in [ReadConnectionless,	ignore		ABORTING
62	ABORTING	M_SendFres_cnf		ignore		ABORTING
63	ABORTING	M_SendAck_cnf		ignore		ABORTING
64	ABORTING	TimerExpired (FragTimeout)	KeepOpNum in [ReadConnectionless,	/* Note: do not keep response any longer */ M_do_halfBound		IDLE
65	ABORTING	TimerExpired (FragTimeout)	KeepOpNum not-in [ReadConnectionless,	ignore		ABORTING
66	ABORTING	RSI_Error_ind		ignore		ABORTING
67	ABORTING	RSI_Abort.req ()			M_SendError_req	ABORTING

Name	Type	Meaning
FragTimeout	Timer	Timeout for partial request detection; Value: 4 * RSI-I::FragTimeout
StartTimer	Function	This function is used to start or restart a timer.
StopTimer	Function	This function is used to stop a timer.
TimerExpired	Function	This function signals that a timer is expired.
rsap	Value	This RSI-Responder, refer to RSI-LMPM::Rsi-List
SendSeqNum	Value	The largest SendSeqNum sent
SendWindowSize	Value	The WindowSize of this RSI-Responder
RecvAckNum	Value	The largest AckSeqNum received
RecvSendNum	Value	The largest SendSeqNum reveiced
RecvFragOffset	Value	The next FragOffset to receive, offset of CallBuffer
RecvWindowSize	Value	The tailored WindowSize of the RSI-Initiator /* The WindowSize within the first Connect fragment will be taken, see M_UpdateAckInfo */
SendArray	Value	Array of 1 to n response fragments to send. SendArray.Length is the count of fragments fragments ::= (sendSeqNum, sendTack, sendMoreFrag, offset, dataLen, Data[])
MaxVarPartLen	Const	The constant value 1432, see RTA-PDU::VarPartLen
M_do_fullyBound	Macro	(rsap.IMAC, rsap.ISAP, rsap.ARUUID) := (PDU.SourceMac, PDU.SourceSAP,
M_do_halfBound	Macro	(rsap.IMAC, rsap.ISAP, rsap.ARUUID) := (undef, undef, undef)
isFirstFrag	Condition	PDUType.Type == RTA_TYPE_FREQ && PDU.Foffset == 0
isLastFrag	Condition	PDUType.Type == RTA_TYPE_FREQ && PDU.AddFlags.MoreFrag == 0
PDUOffset	Macro	// The fragmented part of the PROFINETIOServiceReqPDU starts at this Offset within the RSI-PDU
KeepOpNum	Value	Keep PDU Data
KeepArgsRspMax	Value	Keep PDU Data
KeepCallSequence	Value	Keep PDU Data

M_MoreFreqCheck	Macro	// outsourced checks to keep overview
		if (isFirstFrag)
		   KeepOpNum := PDU.FOpNum
		   KeepArgsRspMax := PDU.ResMaxLength
		   KeepCallSequence := PDU.FCallSequence
		else
		   if (KeepOpNum != PDU.FOpNum) return false
		endif
		if ((PDU.VarPartLen + RecvFragOffset) > CallBuffer.MaxLength) return false
		return true

M_Copy_FragData	Macro	Copy VarPartLen data from the FREQ-RTA-PDU to the CallBuffer at FOffset

M_UpdateAckInfo	Macro	if (RecvWindowSize == 0)
		    RecvWindowSize := PDU.AddFlags.WindowSize
		    /* Decision: is set with first ACK | FRAG and is no longer changed */
		
		/* Update */
		if (PDU.AckSeqNum > RecvAckNum)
		    RecvAckNum := PDU.AckSeqNum

build	Macro	Marshal a RSI-RES-PDU from it's arguments

M_PrepareResponse	Macro	// Prepare the response from RSI_Call.rsp to an array of FRAG-RTA-PDU to send
		RSI-RES-PDU Data := build (PNIOStatus, ArgsRspLength, ArgsRsp)
		DataLen := sizeOf (Data)
		FragPartLen := MaxVarPartLen - 4/*sizeOf FOpnumOffset*/
		
		SendArray := []
		ssn := SendSeqNum;
		offset := 0
		for (len := 0, i := 0; len < DataLen; len += FragPartLen, i += 1)
		  ssn += 1
		  if (len + FragPartLen < DataLen)
		    sendTack := ((i+1) % RecvWindowSize) == 0 ? 1 : 0
		    SendArray[i] := (ssn, sendTack, sendMoreFrag := 1, offset,
		                  FragPartLen, Data[len .. len + FragPartLen - 1])
		    offset += FragPartLen
		  else
		    rest := DataLen - len
		    SendArray[i] := (ssn, sendTack := 0, sendMoreFrag := 0, offset,
		                  rest, Data[len .. len + rest - 1])
		  endif
		endfor

M_PrepareReSend	Macro	// Received an Ack, prepare to ReSend the array of FRAG-RTA-PDU
		
		for (i := 0; i < SendArray.Length; i += 1)
		  if (SendArray[i].SendSeqNum >= RecvAckNum)
		    return SendArray[i].SendSeqNum
		endfor
		return SendSeqNum

M_CheckMoreToSend	Macro	// check if there is more to send
		// returns:
		// - Send: send next FRAG of prepared array
		// - WaitAck: wait for ACK, has more FRAGs to send or wait for ACK of the last FRAG
		// - LastFragAcked: the last FRAG was acked (implicit or explicit)
		// - LastRIFragSend: the last FRAG with OpNum ReadConnectionless or ReadImplicit was send
		
		idx := SendArray.SearchIndexWithSendSeqNum (SendSeqNum)
		
		if (SendArray[idx].sendMoreFrag == 0) // last frag
		  if (SendArray[i].SendSeqNum <= RecvAckNum) return LastFragAcked
		  if (KeepOpNum in [ReadConnectionless, ReadImplicit]) return LastRIFragSend
		  return WaitAck
		
		if (SendArray[i].sendTack == 1) //wait for ack
		  If (SendArray[i].SendSeqNum > RecvAckNum) return WaitAck
		  return Send
		
		return Send

SearchIndexWithSendSeqNum	Function	// return the index within SendArray
		
		for (idx := 0; idx < SendArray.Length; ++idx)
		  if (SendArray[idx].SendSeqNum == SendSeqNum) return idx
		endfor
		return undef // shall never happen

Do_Connect_ind	Event	Loval event within state IDLE
Do_LastFrag_ind	Event	Loval event within state DREQ
Do_CheckMoreToSend	Event	Local event within state FRSP

ConnectResponseState	Value	Track the state of the connect indication ConnectResponseState: IDLE, PING, ABORT

M_CheckConnect	Macro	/* calculate behave of RSI_Connect_ind */
		if (ConnectResponseState == ABORT) return ABORT
		if (ConnectResponseState == PING && PDU.SendSeqNr != RecvSeqNum) return ABORT
		return ConnectResponseState

M_SendFres_req	Macro	/* Send a FRES-RTA-PDU */
		idx := SendArray.SearchIndexWithSendSeqNum (SendSeqNum)
		
		PDU.DA := rsap.IMAC
		
		PDU.DestinationSAP := rsap.ISAP
		PDU.SourceSAP := rsap.RSAP
		PDU.PDUType.Version := 2
		PDU.PDUType.Type := RESPONSE
		PDU.AddFlags.WindowSize := SendWindowSize
		PDU.AddFlags.TACK := SendArray[idx].sendTack
		PDU.AddFlags.MoreFrag := SendArray[idx].sendMoreFrag
		PDU.AddFlags.Notification := (RSI-R-Notification.state == NOTIFY) ? 1 : 0
		PDU.SendSeqNum := SendSeqNum
		PDU.AckSeqNum := RecvSendNum
		PDU.VarPartLen := SendArray[idx].dataLen + 4/*sizeOf FOpnumOffset*/
		
		PDU.FOpnumOffset.Offset := SendArray[idx].offset
		PDU.FOpnumOffset.OpNum := KeepOpNum
		PDU.FOpnumOffset.CallSequence := KeepCallSequence
		PDU.Data := SendArray[idx].Data
		
		LMPM_A_Data.req

M_SendFres_cnf	Macro	LMPM_A_Data.cnf /PDU.PDUType.Type == RESPONSE

M_SendAck_req	Macro	/* Send a RTA-ACK-PDU */
		PDU.DA := rsap.IMAC
		
		PDU.DestinationSAP := rsap.ISAP
		PDU.SourceSAP := rsap.RSAP
		PDU.PDUType.Version := 2
		PDU.PDUType.Type := ACK
		PDU.AddFlags.WindowSize := SendWindowSize
		PDU.AddFlags.TACK := 0
		PDU.AddFlags.MoreFrag := 0
		PDU.AddFlags.Notification := (RSI-R-Notification.state == NOTIFY) ? 1 : 0
		PDU.SendSeqNum := SendSeqNum
		PDU.AckSeqNum := RecvSendNum
		PDU.VarPartLen := 0
		
		LMPM_A_Data.req

M_SendAck_cnf	Macro	LMPM_A_Data.cnf /PDU.PDUType.Type == ACK

Appendix E.

State Machine of the Advanced Variant

#	Current State	Event	Condition	Actions taken	Output	Next State
1	RSI	AlarmSend.req(High)		InsertInArray set AlarmHigh if (FragTimer == running) FragTimerIsRunning = TRUE else FragTimerIsRunning = FALSE StopTimer (FragTimeout) StartTimer (AlarmTimeout)	Do_CheckFreeWindow	ALARMS
2	RSI	AlarmSend.req(Low)		InsertInArray set AlarmLow if (FragTimer == running) FragTimerIsRunning = TRUE else FragTimerIsRunning = FALSE StopTimer (FragTimeout) StartTimer (AlarmTimeout)	Do_CheckFreeWindow	ALARMS
3	RSI	TimerExpired				RSI
4	RSI	Alarm.ind	PDU.SendSeqNum == (RecvSeqNum + 1)	RecvSeqNum := PDU.SendSeqNum	UpperAlarm.ind M_SendAck_req	RSI
	RSI	Alarm.ind	PDU.SendSeqNum != (RecvSeqNum + 1)		M_SendAck_req	RSI
6	ALARMS	Do_CheckFreeWindow	M_CheckFreeWindow == True		M_SendRSI_req	ALARMS
7	ALARMS	Do_CheckFreeWindow	M_CheckFreeWindow == False			ALARMS

8	ALARMS	RSI_Ack.ind	PDU.SendSeqNum == (RecvSeqNum + 1)
			&& M_AlarmsInSendArray == False
				KeepSendSeqNum = PDU.SendSeqNum
				SendSeqNum++
				StopTimer(AlarmTimeout)
					if (FragTimerIsRunning == TRUE)
					StartTimer(FragTimeout)
						RSI

	ALARMS	RSI_Ack.ind	PDU.SendSeqNum == (RecvSeqNum + 1) && M_AlarmsInSendArray == True	KeepSendSeqNum = PDU.SendSeqNum SendSeqNum++ StopTimer (AlarmTimeout) StartTimer (AlarmTimeout)	M_SendRSI_req	ALARMS
10	ALARMS	TimerExpired	TimeoutCount <= MaxFragTimeoutCount	TimeoutCount += 1	M_SendFreq_req	ALARMS

11	ALARMS	TimerExpired (AlarmTimeout)	TimeoutCount > MaxFragTimeoutCount		StopTimer(AlarmTimeout)
					if (FragTimerIsRunning == TRUE)
					StartTimer(FragTimeout)
					TimerExpired (FragTimeout)
						RSI

	ALARMS	AlarmSend.req(Low)	AlarmHigh is empty	InsertInArray set AlarmLow		ALARMS
13	ALARMS	AlarmSend.req(High)	AlarmHigh is empty	InsertInArray set AlarmHigh		ALARMS
14	ALARMS	AlarmSend.req(High)	AlarmHigh is full		AlarmSend.cnf(-)	ALARMS
15	ALARMS	AlarmSend.req(Low)	AlarmLow is full		AlarmSend.cnf(-)	ALARMS
16	ALARMS	M_SendFreq_cnf		ignore		ALARMS
17	ALARMS	RSI_Abort.req			/* forward request RSI_Abort.req()	RSI
18	ALARMS	Alarm.ind	PDU.SendSeqNum == (RecvSeqNum + 1)	RecvSeqNum := PDU.SendSeqNum	UpperAlarm.ind M_SendAck_req	ALARMS
19	ALARMS	Alarm.ind	PDU.SendSeqNum != (RecvSeqNum + 1)		M_SendAck_req	ALARMS
20	ALARMS	RSI Events	Not defined	Put in Queue		ALARMS

Name	Type	Meaning
AlarmHigh	Value	Local variable of alarm high to send (sendSeqNum, sentAlready)
AlarmLow	Value	Local variable of alarm low to send (sendSeqNum, sentAlready)
Do_CheckFreeWindow	Event	Signal an internal Event with Trigger as Parameter. This Parameter is used by M_CheckFreeWindow()

M_CheckFreeWindow	Makro	idx := SendArray.SearchIndexWithSendSeqNum (SendSeqNum)
		
		    if (SendArray[idx].sendMoreFrag == 0) return WaitAck
		    if (SendArray[idx].sendTack == 1) return WaitAck
		
		if (SendSeqNum - KeepSendSeqNum == 2
		    && SendArray[idx].sendMoreFrag == 1
		    && SendArray[idx].sendTack == 1)
		return FALSE
		else
		return TRUE

Appendix F.

LMPM State Machine

The LMPM state machine is the connection between the initiator and responder state machines to the Ethernet Device Driver state machine. It is designed by Thomas Pössler. (Pössler, 2018).

#	Current State	Event	Condition	Actions taken	Output	Next State
1	IDLE	LMPM_A_Data.ind	CheckRSI() == Ignore	ignore		IDLE
2	IDLE	LMPM_A_Data.ind	CheckRSI() == RespondError(err)	PNIOStatus := err	M_Tx_Error_req	IDLE
3	IDLE	M_Tx_Error_cnf		ignore		IDLE
4	IDLE	LMPM_A_Data.ind	CheckRSI() == IndicateError(err, sap)	/* sap may be rsap or isap */ PNIOStatus := err	RSI_Error_ind	IDLE
5	IDLE	LMPM_A_Data.ind	CheckRSI() == Ack	sap := Rsi-DsapLookup /* may be rsap or isap */	RSI_Ack_ind /* explicit Ack */	IDLE
6	IDLE	LMPM_A_Data.ind	CheckRSI() == Freq	rsap := Rsi-DsapLookup	RSI_Ack_ind /* implicit Ack */ RSI_Freq_ind	IDLE
7	IDLE	LMPM_A_Data.ind	CheckRSI() == Fres	isap := Rsi-DsapLookup	RSI_Ack_ind /* implicit Ack */ RSI_Fres_ind	IDLE
8	IDLE	LMPM_A_Data.ind	CheckRSI() == ConnectFreq	rsap := Rsi-Alloc	RSI_Connect_ind RSI_Ack_ind RSI_Freq_ind	IDLE
9	IDLE	LMPM_A_Data.ind	CheckRSI() == ConnectReRun(rsap)	/* use existing rsap */	RSI_Connect_ind RSI_Ack_ind RSI_Freq_ind	IDLE
10	IDLE	RSI_Abort.req		Rsi-Free (sap)		IDLE
11	IDLE	RSI_R_Add.req		Add RSI-R-Interface to RSI-List	RSI_R_Add.cnf	IDLE
12	IDLE	RSI_R_Remove.req		Remove RSI-R-Interface from RSI-List	RSI_R_Remove.cnf	IDLE
13	IDLE	RSI_I_Add.req		Add RSI-I-Interface to RSI-List	RSI_I_Add.cnf	IDLE
14	IDLE	RSI_I_Remove.req		Remove RSI-I-Interface from RSI-List	RSI_I_Remove.cnf	IDLE

Name	Type	Meaning

CheckRSI	Function	//possible returns:
		//- Ignore
		//- ConnectFreq
		//- ConnectReRun
		//- Freq
		//- Fres
		//- Ack
		//- IndicateError(err)
		//- RespondError(err)
		
		if (! Valid-RTA-PDU) return Ignore
		if (! Valid-RSI-PDU) return Ignore
		PDU := A_SDU
		
		sap := Rsi-DsapLookup ()
		if (sap == undef) return CheckFreqConnect ()
		
		if (PDU.PDUType == FREQ) return CheckFReqData ()
		if (PDU.PDUType == FRES) return CheckFResData ()
		if (PDU.PDUType == ACK) return CheckAckData ()
		if (PDU.PDUType == ERROR) return CheckErrorData ()
		return Ignore

Valid-RTA-PDU	Function	if (EthType != 0x8892) return false
		if (FrameID != 0xFE02) return false
		if (DestinationMac != InterfaceMac) return false
		if (SourceMac == Multicast) return false
		
		if (DestinationSAP not-in [0..0x7FFF,0xFFFF]) return false
		if (SourceSAP not-in [0..0x7FFF]) return false
		if (PDUType.Type not-in [ACK,ERR,FREQ,FRES]) return false
		if (PDUType.Version != 2) return false
		if (AddFlags.WindowSize not-in [2..7]) return false
		if (AddFlags.<Rest>) # do not check
		if (SendSeqNum not-in [0..0x7FFF,0xFFFE]) return false
		if (AckSeqNum not-in [0..0x7FFF,0xFFFE]) return false
		if (VarPartLen not-in [0..0x0598]) return false
		
		return true

Valid-RSI-PDU	Function	if (PDUType in [ACK, ERR]) return true /* check later */
		
		if (PDUType == FREQ)
		     if (FOpnum in [0,2..8]
		     && FOffset in [0..0x00FFFFFF]
		     && FCallSequence # do not check
		     && RSI-XXX-PDU # do not check
		     )
		        if (ResMaxLength in [4 .. 0x00FFFFFF]) return true
		
		if (PDUType == FRES)
		     if (FOpnum in [0,2..8]
		     && FOffset in[0..0x00FFFFFF]
		     && FCallSequence # do not check
		     && PNIOStatus # do not check
		     && RSI-XXX-PDU # do not check
		     ) return true
		
		return false

CheckFReqConnect	Function	if (PDU.DestinationSAP != CON-SAP) return Ignore
		if (PDU.PDUType != FREQ) return Ignore
		if (PDU.SendSeqNum not-in [0..0x7FFF) return Ignore
		if (PDU.AckSeqNum != 0xFFFE) return Ignore
		if (PDU.FOffset != 0) return Ignore
		
		if (FOpNum == Connect or FOpNum == ReadConnectionless)
		    if ((rsap := Rsi-IsapLookup(AR)) != undef) return ConnectReRun(rsap)
		    if (! Rsi-InterfaceAvailable(AR)) return RespondError(InterfaceNotFound)
		    if (! Rsi-AllocPossible(AR)) return RespondError(OutOfARRessource)
		    return ConnectFreq
		
		if (FOpNum == ReadImplicit)
		    if ((rsap := Rsi-IsapLookup(RI)) != undef) return ConnectReRun(rsap)
		    if (! Rsi-InterfaceAvailable(RI)) return RespondError(InterfaceNotFound)
		    if (! Rsi-AllocPossible(RI)) return RespondError(OutOfARRessource)
		    return ConnectFreq
		
		return Ignore

CheckFReqData	Function	if (PDU.SA != SourceMAC) return Ignore
		if (PDU.SourceSAP != SourceSAP) return Ignore
		if (PDU.AckSeqNum > SendSeqNum) return Ignore
		
		return Freq

CheckFResData	Function	if (PDU.SA != SourceMAC) return Ignore
		if (PDU.SourceSAP != SourceSAP) return Ignore
		if (PDU.AckSeqNum > SendSeqNum) return Ignore
		
		return Fres

CheckAckData	Function	if (PDU.SA != SourceMAC) return Ignore
		if (PDU.SourceSAP != SourceSAP) return Ignore
		if (PDU.AckSeqNum > SendSeqNum) return Ignore
		
		return Ack

CheckErrorData	Function	if (PDU.SA != SourceMAC) return Ignore
		if (PDU.SourceSAP != SourceSAP) return Ignore
		if (PDU.PNIOStatus == 0) return Ignore
		
		return IndicateError(PNIOStatus, sap)

sap	Value	Means rsap or isap
rsap	Value	Responder-SAP Values refer to entry within RSI-List or "undef" if not contained.
isap	Value	Initiator-SAP Values refer to entry within RSI-List or "undef" if not contained.

RSI-List	Value	List of Union (
		  (Typ(ANY): unique-key: SAP,
		    State, SourceMAC, SourceSAP, VendorID, DeviceID, Instance)
		  (Type(R): unique-key: RSAP,
		    State, IMAC, ISAP, VendorID, DeviceID, Instance, Binding, CallBuffer)
		  (Type(I): unique-key: ISAP,
		    State, RMAC, RSAP, VendorID, DeviceID, Instance, Binding)
		)

Rsi-InterfaceAvailable(B_)	Function	True if the tuple ("R", Binding(B_), VendorID,DeviceID,Instance) is in Rsi-List
Rsi-AllocPossible(B_)	Function	Looking for a free entry within RSI-List: "R", Binding(B_), VendorID, DeviceID, Instance must match Use rsap if Binding is AR and IMAC and ISAP matches and the state of the instance is IDLE or the state if the instance is ABORTING and KeepOpNum is ReadConnectionless. Use rsap if the state of the instance is IDLE. Use rsap if the state of the instance is ABORTING and KeepOpNum is ReadImplicit. return true if there is a entry with state free
Rsi-Alloc	Function	Alloc the free entry within RSI-List found by Rsi-AllocPossible(): Mark found rsap's state as used return rsap
Rsi-Free	Function	Mark sap's state as free

Rsi-DsapLookup	Function	sap := Search PDU.DestinationSAP within RSI-List
		isap := rsap := undef
		if (sap found)
		if (Typ == "R") rsap := sap
		if (Typ == "I") isap := sap
		return sap
		return undef

Rsi-IsapLookup(B_)	Function	Search IMAC, ISAP with in RSI-List with Type R and Binding B_. Return rsap if found. Return undef if not found.
M_Tx_Error_req	Macro	Send an ERR-RTA-PDU back to originator SourceMAC, SourceSAP
FOpNum	Macro	PDU.FOpnumOffset.Opnum
FOffset	Macro	PDU.FOpnumOffset.Offset
FCallSequence	Macro	PDU.FOpnumOffset.CallSequence
>	Operator	Compares two SeqNum, take care of wrap around For valid range see definition of SendSeqNum and AckSeqNum

Appendix G.

Comparison Fragmentation

Protokoll	**TCP**	**IPv4**	**IPv6**	**IKEv2**	**SCTP**	**RPC**
Bits with Frag	11	21	21	21	11	17
Bits without Frag	11	21	1	9	1	17
	SEQ-Number 8	Identification: 8	Next Header: 1	Message ID: 8	Chunk Type: 1	Flags: 3
	ACK, SYN-FIN Flag: 3	Offset 11	Fragment Offset: 11	Flag Header: 1	TSN: 8	Serial: 8
		Flags DF + MF: 2	More Flag: 1	Maximum Fragmente: 47: 6	Flags: 2	Fragment Number: 6
			Identifiaction: 8	MaxTotal Fragments: 47: 6		

Comparison Conditions	
Max Payload Size	1432Byte
Max Data Length	54KByte
Uniqueness of one packet	255 (8 Bit)

Appendix H.

Comparison Memory

Sequence Numbers / Uniqueness	32Bit	
WindowSize	8	
Length	65546 Byte	
MTU	1400	-> is known

TCP	Bits	IPv4	Bits	
WindowSize	8	Identification	32	
SND.UNA	32	FO	10	
SND.NXT	32	MF	1	
SND.WND	4	NFB	6	
SND.UP	32	Timer	192	
SND.WL1	32	OFO	10	Only Send
SND.WL2	32	OMF	1	
ISS	32	OTL	32	
RCV.NXT	32	OIHL	4	
RCV.WND	4			
RCV.UP	32			Only Recv
IRS	32	RCVBT	1024	
SEG.SEQ	32	Total Length	32	
SEG.ACK	32			
SEG.LEN	4			
SEG.WND	4			
SEG.UP	32			
user_calls(OPEN, SEND, RECEIVE, CLOSE, ABORT, and STATUS)	-			
States: LISTEN, SYN-SENT, SYN-RECEIVED, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT, and the fictional state CLOSED	10			
Timer	192			
	610	Recv	1344	
		Send	1344	

IPv6	Bits			IKEv2	Bits	
Identification	32			Message ID	32	
Next Header	4					
MF	1					
Timer	192					
RCVBT	1024		not declared, assumption	Total Fragments	6	Only Recv
PL.orig	32			Fragment Bitmask	46	
PL.first	6	Only Recv				
FL.first	6			ICV (AES_GCM)		withouth Integrety Check
FO.last	6					
FL.last	6					
Fragment Offset	10	Only Send		Fragments Sent	6	Only Send
Total Length	32			Number Fragments	6	
				Total Length	32	
				Timer	192	
Recv	1351			Recv	84	
Send	1351			Send	268	

SCTP	Bits	RPC	Bits
		Sequence Number	32
		window_size	8
Base TSN	32	Timer	192
a rnd	8		
Timer	192		
Number Gaps	3		
Received / Gaps	32	Received / Gaps	8
Base ACK	32	serial_num	32
Dublicated	Not compared	Dublicated	Not compared
Counter Received	4	number selack	1
		selack	8
Next TSN to Send	32		
		Next Sequence Number to send	32
Receive	335		313
Send	335		313