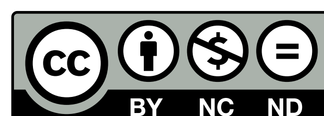
THE BOOK OF SCHC

Paul OULTRY

IMT ATLANTIQUE

This work is licensed under a Creative Commons “Attribution-NonCommercial-NoDerivs 3.0 Unported” license.



Published 30 octobre 2024

SCHOOL

Table of Contents

1 Introduction 8

1.1 Compression mechanisms 8

1.2 Internet of Things 9

1.3 The Book of SCHC 11

2 Internet protocol stack 12

2.1 Introduction 12

2.2 IPv6 14

2.3 Wireshark and pcap 15

2.4 Scapy 16

2.5 UDP 18

2.6 ICMPv6 18

2.7 CoAP 18

2.7.1 REST principle 19

2.7.2 CoAP Header 20

2.7.3 Reliable Messages 21

2.7.4 Token 23

2.7.5 Options 25

2.7.6 CoAP Extensions 25

2.8 Hand-on CoAP 31

3 SCHC architecture elements . . . 34
3.1 Rule ID's 35
3.2 Rules structure 37
3.3 The Rule Manager 38
3.3.1 Adding Rules into the Rule Manager . . . 39
3.3.2 Adding a Set of Rules . . . 40
3.3.3 Attributing a Set of Rules to a Device . . . 40
3.3.4 Importing rules from a JSON file . . . 41
3.4 Exercises 42
4 Compression . . . 43
4.1 Compression rule 43
4.1.1 Field identification. . . . 44
4.1.2 Field Matching. . . . 46
4.1.3 Compression . . . 47
4.1.4 Decompression . . . 48
4.2 Simple compression rules for openSCHC. 48
4.3 Hands-on 51
4.3.1 Parsing . . . 51
4.3.2 Rule Manager . . . 54
4.3.3 Rule Matching . . . 55
4.4 Hands-on 57
4.4.1 Let's compress . . . 57
4.4.2 Let's decompress . . . 59
4.5 Destructive compression 62
4.5.1 Hop Limit . . . 63
4.5.2 Flow Label . . . 63
4.6 Rule description Optimization 63
4.7 CoAP Rules 64
4.7.1 Option description . . . 64
4.7.2 Variable Length . . . 64
4.8 Hand-on 65
5 Networking . . . 68
5.1 Running the scenario 69
5.2 First ping 69

5.3 Compressing traffic on *Core* 70
5.3.1 Get ping request packets 70
5.3.2 Generating rule for ICMPv6 Ping Request 72
5.3.3 The SCHC Machine 75
5.3.4 On the Device 78
5.3.5 Back to the Core 81

6 Simple CoAP 84

6.1 Network architecture 84

6.2 LPWAN scenario 85
6.2.1 The Rules 86
6.2.2 On the device 88
6.2.3 The decompression 89
6.2.4 aiocoap 90
6.2.5 Returning errors 92
6.2.6 Processing errors on the Device 92

6.3 Underwater scenario 94
6.3.1 The basic rules 94
6.3.2 The application client 95
6.3.3 The device server 96
6.3.4 Several Optimization 98

7 Fragmentation 99

7.1 Rules 100

7.2 No Ack 101
7.2.1 Padding management 103
7.2.2 Impact of FCN size 104
7.2.3 Conclusion 105

7.3 Hands-on No Ack 106

7.4 Dtag 106

7.5 Ack Always 106
7.5.1 Sliding Window 107
7.5.2 Missing fragment 108
7.5.3 Bitmap Request 108
7.5.4 Terminating an exchange 109
7.5.5 Byte borrowing 110
7.5.6 Conclusion 110

7.6 Ack on Error **111**
7.6.1 Variable MTU and Tiles 112

8 Ping fragmentation **116**

8.1 Set up the environment **116**

8.2 Large pings **120**
8.2.1 NoAck rule 120

9 Conclusion **123**

10 Answers to the questions **125**

11 OpenSCHC Identifiers **144**

11.1 Field Id **144**

Acronyms

3GPP 3rd Generation Partnership Project
6LOWPAN IPv6 Low power Wireless Personal Area Networks
AS Application Server
CBOR Concise Binaire Object Representation
CDA Compression Decompression Action
CoAP Constrained Application Protocol
CRTP RTP Header Compression
CTCP TCP Header Compression
DI Direction Indicator
ECN Explicit Congestion Notification
FCN Fragment Compressed Number
FID Field IDentifier
FL Field Length
FV Field Value
ICMP Internet Control Message Protocol
IID Interface IDentifier
IP Internet Protocol
IPv4 Internet Protocol version 4
IPv6 Internet Protocol version 6
LPWAN Low Power Wide Area Network
LNS LoRaWAN Network Server
LSB Less Significant Bit
MO Matching Operator
MSB Most Significant Bit
MTU Maximum Transmission Unit
NDP Neigbor Discovery Protocol
RCS Reassembly Check Sequence
ROHC Robust Header Compression
SCHC Static Context Header Compression
TCP Transmission Control Protocol
TV Target Value
UDP User Datagram Protocol
URI Universal Resource Identifier

1. Introduction

Youtube

This book serves as the companion text to the Coursera MOOC *SCHC : A new era of interoperability*. It provides a detailed exploration of SCHC, a groundbreaking technology designed to optimize communications in Internet of Things (IoT) networks. The book combines theoretical foundations with practical implementations, offering readers both the understanding and hands-on experience needed to master SCHC.

1.1 Compression mechanisms

The transformative power of compression technologies extends far beyond their technical implementations – they have historically been catalysts for revolutionary changes in how we communicate. Consider the early days of Internet connectivity : without the advent of modem compression techniques, the dream of bringing internet access to millions of homes through standard telephone lines would have remained just that – a dream. The ability to compress data made it possible to squeeze more information through the limited bandwidth of copper wires, directly enabling the massive expansion of internet adoption in the 1990s.

This pattern repeated itself during another crucial transition in telecommunications history : the shift from circuit-switched to packet-switched networks for voice communications. Voice over IP (VoIP) became practically viable only when compression technologies

evolved enough to reduce voice data packets to manageable sizes while maintaining call quality. This compression capability proved instrumental in enabling the telecommunications industry's transition from traditional circuit-switched telephony to modern packet-based cellular networks.

In the ever-expanding landscape of connected devices, one of the most critical challenges has been the efficient transmission of data across constrained networks. Although Internet Protocol (IP) has served as the backbone of network communications for decades, its overhead poses significant challenges when deployed in Internet of Things (IoT) scenarios. This is where the story of header compression, particularly Static Context Header Compression (SCHC), begins to unfold.

Today, we see the same pattern emerging with SCHC in the IoT landscape : Just as compression made the Internet and VoIP accessible and practical, SCHC is making IP-based communications viable for resource-constrained IoT devices, catalyzing another wave of technological transformation. This historical perspective reinforces a crucial lesson : compression technologies don't just optimize existing systems ; they enable entirely new possibilities and drive paradigm shifts in how we build and use communication networks.

1.2 Internet of Things

The development toward efficient header compression in IoT networks reflects a recurring pattern in technological evolution. In the early 2010s, the IoT landscape was marked by fragmentation and competing technologies. Each new communication protocol that emerged promised to revolutionize the industry in its own way. LoRaWAN carved out its niche by enabling long-range communications with minimal power consumption, while SigFox pioneered ultra-narrow-band transmissions for specific use cases. Cellular IoT solutions offered the advantage of widespread coverage, though at the cost of higher power requirements.

This diversity of approaches, while innovative, created significant challenges for developers and system architects. Each new application required custom solutions, leading to a cascade of complications. Development costs soared as teams needed to create and maintain multiple implementations of protocols. The time to market stretched longer as each new deployment required extensive customization. Perhaps most critically, interoperability became a significant hurdle, with devices using different protocols unable to communicate effectively with each other or existing infrastructure.

The Internet Protocol had already proven itself as a solution to similar interoperability challenges in traditional networking. However, as explained earlier, when applied to IoT scenarios, IP revealed new challenges. A typical IPv6 packet header alone consumes 40 bytes, followed by additional protocol headers such as UDP with 8 bytes. For an IoT device sending just a few bytes of sensor data, this overhead could be more than ten times the

size of the actual payload. In the resource-constrained world of IoT, such inefficiency was simply untenable.

The challenges facing IoT deployments were multifaceted and interconnected. Energy efficiency has become a primary concern, particularly evident in the case of smart energy meters. These devices, designed to operate for years without maintenance, found themselves consuming up to 70% of their energy by simply transmitting data. Without optimization, devices that should have been operating for years might require monthly battery changes, making large-scale deployments impractical.

The impact of SCHC becomes particularly evident in smart metering deployments. Traditional implementations requiring 200-byte daily transmissions with battery life of two to three years have been transformed. With SCHC, these same devices can operate with just 20-30 bytes of daily transmission, extending battery life beyond ten years and enabling deployment across multiple network types. The technology has proven especially valuable for multi-radio devices, where SCHC serves as a common IP interface layer sitting between the application and various radio options.

Bandwidth restrictions further complicated the situation. LoRaWAN networks impose strict limitations on data transmission, with maximum payloads ranging from 51 to 242 bytes and daily transmission quotas of approximately 30KB. Satellite communications faced similar constraints, with the added burden of high costs per transmitted byte and limited transmission windows.

Static Context Header Compression emerged as an elegant solution to these challenges. Rather than transmitting complete headers with every packet, SCHC takes advantage of the predictable nature of IoT communications. Most IoT devices communicate with the same endpoints using consistent patterns, allowing significant optimization through context-aware compression. This approach has transformed the landscape of IoT communications, reducing 40-byte headers to mere bits while maintaining full functionality.

Security considerations have been paramount in the development of SCHC. The technology achieves its efficiency without compromising security through a sophisticated multi-layer approach. By first compressing application data, then applying encryption through DTLS, and finally compressing the encrypted payload, SCHC achieves an 80% reduction in transmitted bytes while maintaining support for robust 4K certificates and full security compliance.

The future applications of SCHC extend beyond traditional IoT scenarios. Ambient IoT represents an exciting frontier, where battery-free sensors and energy harvesting devices communicate through 5G networks with RFID-like efficiency. Even more ambitious applications include interplanetary communications, where efficient header compression could play a crucial role in space-to-Earth communications.

1.3 The Book of SCHC

Looking forward, the evolution of header compression in IoT represents more than just technical optimization—it marks a fundamental shift in how we approach efficient communications in constrained environments. As we explore the technical depths of these solutions in subsequent chapters, we'll see how proper implementation of header compression techniques can transform the possibilities for IoT deployments, enabling new applications while optimizing existing ones.

The following chapters will delve into the technical implementation details of SCHC, exploring various deployment scenarios and examining real-world case studies. We will investigate the nuances of performance optimization and discuss emerging applications that push the boundaries of what's possible in IoT communications. Through this exploration, readers will gain a comprehensive understanding of how header compression is shaping the future of connected devices and enabling the next generation of IoT innovations.

Chapter 1 presents the foundations and history of header compression in IoT networks. It explains how compression has enabled major technological transitions in telecommunications. It shows why SCHC was needed for IoT devices.

Chapter 2 introduces the Internet protocol stack essential for SCHC. It explains IPv6, UDP, and CoAP protocols. It describes how to analyze network traffic with Wireshark and Scapy tools.

Chapter 3 describes the elements of the SCHC architecture. It details rule IDs and their organization. It explains how the Rule Manager works. It shows how to configure rules for different devices.

Chapter 4 focuses on SCHC compression. It explains field identification and matching operations. It details compression and decompression actions. It provides examples with IPv6 and UDP headers. It introduces CoAP header compression.

Chapter 5 demonstrates networking with SCHC. Shows how to setup a basic topology with the device, the core, and the application. It explains how to compress and route ICMP ping messages. It includes real implementation examples.

Chapter 6 explores CoAP implementations. It presents two scenarios : an LPWAN uplink-focused case and an underwater bidirectional case. It shows how to optimize rules for different network constraints.

Chapter 7 details SCHC fragmentation. It explains the three fragmentation modes : No-Ack, Ack-Always, and Ack-on-Error. It describes how to handle large messages on constrained networks. It shows tile-based fragmentation.

Chapter 8 provides a complete example with ping fragmentation. It combines compression and fragmentation techniques. It shows how to implement bidirectional communication.

Chapter 9 contains answers to all practice questions. It provides detailed explanations for all exercises. It helps verify understanding of core concepts.

2. Internet protocol stack

2.1 Introduction

Most of the network protocols are based on a similar format : a header followed by data and sometime ending with a trailer. Protocols are often stacked, and the data are in fact other protocols coming from the adjacent upper layer as represented in Figure 2.1. Note that the header can be larger than the data.

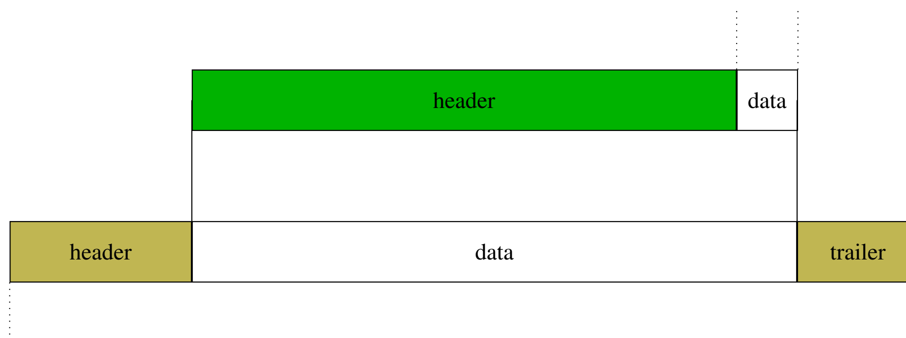


FIGURE 2.1 – Stacked protocols

Following this architecture, the headers from different protocols are contiguous. SCHC may use this property to process several layers at the same time.

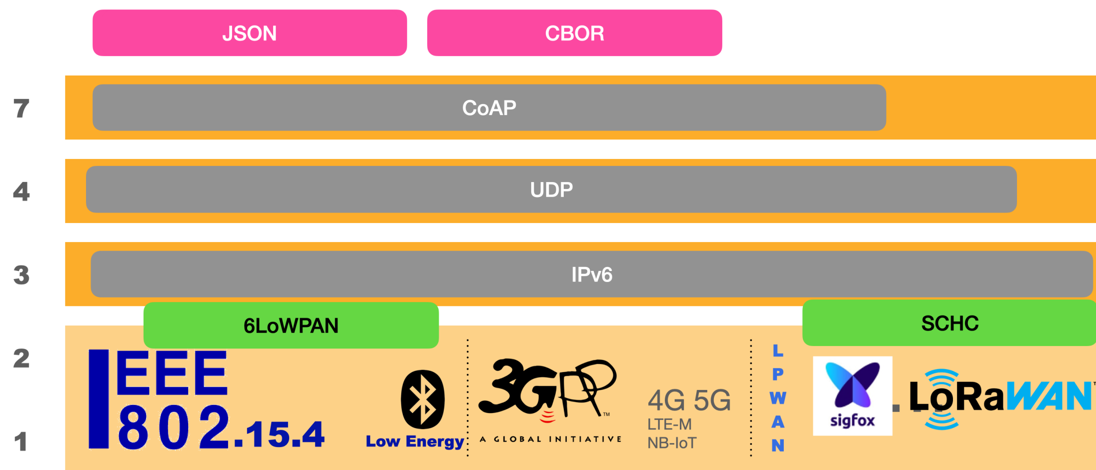


FIGURE 2.2 – ISO Reference Model for IoT

The reference model issued based on 7 layers is widely recognized to formalize network architecture, even if the internet architecture, in a sense, simplified it with only four layers (cf. figure 2.2) :

— An interface layer, composed of the access technology, like Ethernet, Wi-Fi,. . . They form Layer 1 and 2 of the ISO reference model.

— The IP layer, which ensures the forwarding of the information. There are two versions of the IP protocol. IPv4 is the legacy one and is still popular in computer exchanges. IPv6 is supposed to succeed IPv4 in any domain, but the large base of IPv4 services makes the move difficult. For new services, and IoT belongs to that category, IPv6 is the default protocol, and we will base our example on it. This is layer 3 of the OSI Reference Model.

IPv4

IPv6

— Internet Control Message Protocol (ICMP) is associated with IP to control the way the IP layer behaves. Even if ICMP messages are encapsulated in IP packets, ICMP is usually considered a layer 3 protocol. ICMP is also the support for some applications, such as *Ping*. In IPv6, ICMPv6 is used to configure the nodes during bootstrap through Neigbor Discovery Protocol (NDP).

ICMP

— The transport layer includes two protocols :
  — Transmission Control Protocol (TCP) controlling end-to-end the good behavior of the exchange. This is done by establishing a state at each end. The state evolves with the bytes sent and received.
  — User Datagram Protocol (UDP) which provides no control over the exchange

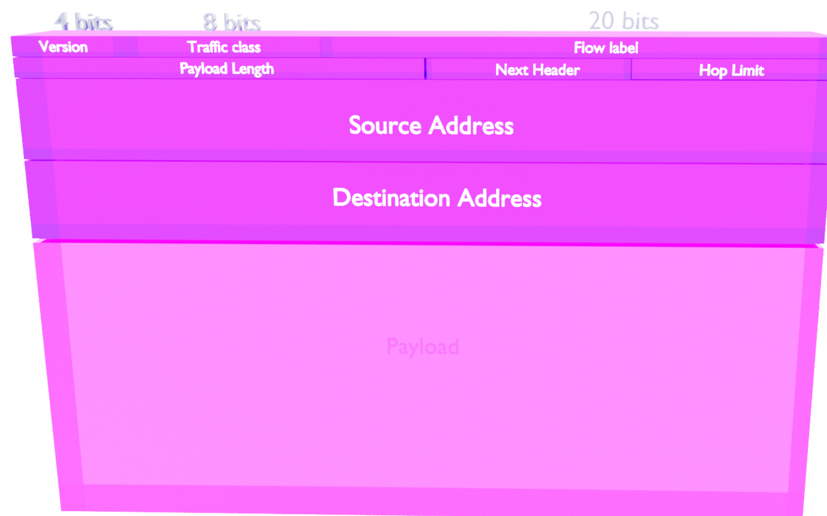


FIGURE 2.3 – IPv6 Header

and, therefore, is very light. IoT applications are mainly based on this protocol.

— Applications, known as layer 7 in the ISO Reference Model, are directly above layer 4 in the Internet Architecture, creating a gap in layer numbering. CoAP is one of the available application protocols. CoAP is a lightweight implementation of the REST paradigm dedicated to constrained objects, as HTTP is widely used by computers.

An header is composed of fields containing specific information, useful for the protocol itself. Fields are often a binary sequence of the specified size, but the structure of the field depends on the protocol itself, regarding its format or how the information is stored inside.

2.2 IPv6

IPv6 is the new version of the Internet Protocol popularized by IPv4. IPv6 contains fewer fields than IPv4, and they are very stable during a session.

The IPv6 header, defined in RFC 8200, is composed of several fields (cf. figure 2.3) :

Version

- a Version on 4 bits : it always contains the value 6.

Traffic Class

- a Traffic Class on 8 bits. This field contains two subfields :

DiffServ

  - 6 DiffServ bits indicates a priority in routers to select which packets should be dropped in case of congestion (cf. RFC 2474).

ECN

  - 2 Explicit Congestion Notification (ECN) bits used to signal that a congestion is occurring and inform the source to reduce its throughput (cf.RFC 3168).

Flow Label

- a Flow Label on 20 bits. The value is selected by the source and keep constant during a flow ; when the application remains the same on both ends. This is the equivalent of the UDP Source and Destination port which may not be available under some conditions. The value 0 indicates that the flow label is not set.

Payload Length

- a Payload Length on 16 bits. It tells the length of the payload immediately after the IPv6 header [1].

Next Header
ICMPv6

- a Next Header on 8 bits pointing to the upper layer protocols. The values used in IPv4, such as 6 for TCP and 17 for UDP are kept, but ICMPv6 is a brand new protocol with value 58 [2].

Hop Limit

- a Hop Limit on 8 bits counting the number of times the packet has been forwarded. This value is set arbitrarily by the source, large enough to cross the Internet. Each router decreases by 1 the value. When the Hop Limit reaches 0 the packet is discarded. The main goal of this field is to limit the effect of routing loops.

link-local

- source and destination addresses in 128 bits. Approximately, there are two kinds of IPv6 addresses. The link-local addresses start with `FE80::/64`. Global prefixes usually start with the value `2000::/3`. Link-local addresses cannot be routed outside of a link (i.e., a layer 2 technology), while global addresses allow one to join any node on the network. The format remains the same for these two categories ; the first 64 bits represent the prefix allocated to the link, and the last 64 bits are unique on the link.

2.3 Wireshark and pcap

pcap

Wireshark is a powerful tool for analyzing packets. It is possible to directly capture the traffic from an interface or read a file in a pcap format. Figure 2.4 on the next page shows the content of the `trace_coap.pcap` file.

In the top section or window, each packet is displayed in an abstract manner, with the packet number in the file, the relative capture time, the source address and port, the destination address and port, the upper protocol name, the packet length, and a summary of the highest-level protocol, CoAP in this case.

The middle window displays the protocol fields of the frame highlighted in the top window. The figure disassembles the IPv6 header, and the corresponding fields can be

1. In IPv4 the length indicates the full length of the datagram including the header.
2. IPv6 may also this field to point out extensions, but they are out of the scope of this document.

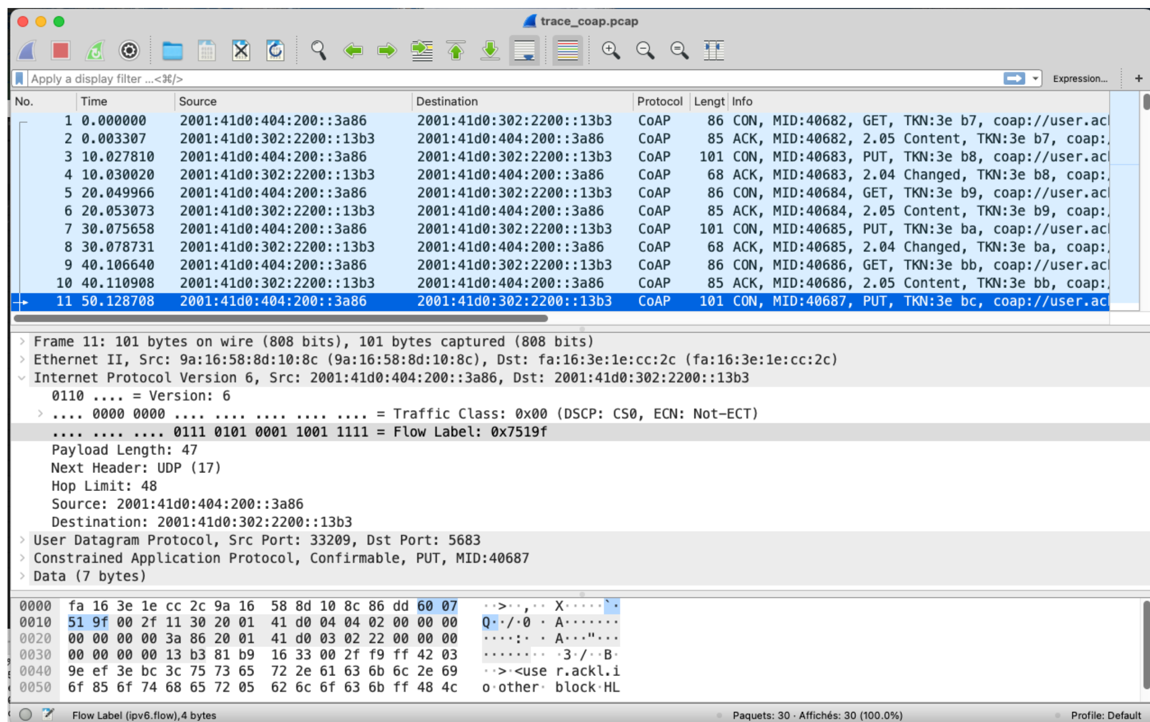


FIGURE 2.4 – Wireshark trace

displayed on the bottom window.

This bottom window displays the packet's content in hexadecimal and ASCII, when a value corresponds to a printable ASCII character.

Question 2.3.1: Packet Length

Launch `wireshark trace_coap.pcap`. Look at packet 6 in the bottom window. Knowing that the first 14 bytes are the Ethernet header, measure the size of the IPv6 packet.
The network analyzer displays a length of 85 bytes, why does the payload length in the header field contain the value 31 ?

2.4 Scapy

Even though Wireshark is a powerful tool for looking at a flow and understanding its content, it is harder to manipulate the packets and establish some statistics. Scapy is a powerful Python module to manipulate packets and even forge new packets. To install scapy, type :

Scapy

```
# pip3 install scapy
```

The following program `pcap_read.py` reads a pcap file and displays its content.

pcap_read.py

Listing 2.1 – pcap_read.py

```python
#!/usr/bin/env python3
from scapy.all import *

# rdpcap comes from scapy and loads in our pcap file
packets = rdpcap('trace_coap.pcap')

for packet in packets:
    packet.show()

    hexdump(packet)

    print ("="*40)
```

— line 2, the scapy module is downloaded [3].

— Line 5, the pcap file is opened.

— Line 7, the loop allows the program to access each packet in the pcal file.

— Lines 8 and 10, each packet is disassembled and displayed in hexadecimal.

The result is as follows.

```
# python3 pcap_read.py | more
###[ Ethernet ]###
  dst       = fa:16:3e:1e:cc:2c
  src       = 9a:16:58:8d:10:8c
  type      = IPv6
###[ IPv6 ]###
     version   = 6
     tc        = 0
     fl        = 479647
     plen      = 32
     nh        = UDP
     hlim      = 48
     src       = 2001:41d0:404:200::3a86
     dst       = 2001:41d0:302:2200::13b3
###[ UDP ]###
        sport     = 33209
        dport     = 5683
        len       = 32
        chksum    = 0x9ca7
###[ Raw ]###
           load      = 'B\x01\\x9e\\xea>\\xb7<user.ackl.io\\x84time'

0000  FA 16 3E 1E CC 2C 9A 16 58 8D 10 8C 86 DD 60 07  ..>..,..X.....`.
0010  51 9F 00 20 11 30 20 01 41 D0 04 04 02 00 00 00  Q.. .0 .A.......
0020  00 00 00 00 3A 86 20 01 41 D0 03 02 22 00 00 00  ....:. .A..."...
0030  00 00 00 00 13 B3 81 B9 16 33 00 20 9C A7 42 01  .........3. ..B.
0040  9E EA 3E B7 3C 75 73 65 72 2E 61 63 6B 6C 2E 69  ..>.<user.ackl.i
0050  6F 84 74 69 6D 65                                o.time
========================================
...
```

Note that layer 7 has not been identified by scapy and is displayed as a raw structure.

3. since the import is done with `from` scapy functions will not be prefixed by `scapy.`

Question 2.4.1: Flow Label

To access to a specific field in a packet, scapy uses the following notation `p[L].f`, where `p` is the Scapy frame, `L` is the layer and `f` is the field in that layer. For example, `packet[UDP].sport` gives access to the UDP source port.
Modify the previous program pcap_read.py to display all the Flow Labels used by IPv6.

2.5 UDP

The UDP header is very simple ; it is composed of four 16-bit-long fields :

— Source port designates the application used by the source,

— Destination port indicates the application that will process the payload,

— Length in bytes of the UDP message. This information is redundant with the IPv6 payload length.

— The checksum is global to the packet and validates the correctness of the IPv6 header, the UDP header, and the payload.

2.6 ICMPv6

ICMPv6 is a control protocol to report errors and performs functions such as diagnostics. Its protocol number is 58, placed on the next header of the IP header format. The header format has the following header fields :

— Type. This field indicates the message type in 8 bits, and its value determines the format of the remaining header.

— Code. This field depends on the Type message value. It is 8 bits long and gives more information about the message type, creating an additional information level.

— Checksum. This field detects data corruption in the ICMPv6 message and part of the IPv6 header. It is 16 bits long.

— Message Body. The size of this field is variable and depends on the message type and code. Each Type provides a format.

2.7 CoAP

CoAP is a lightweight protocol that implements the same REST paradigm as HTTP. Therefore, CoAP works in the client/server mode. The server owns resources, identified by an URI that are manipulated by methods, such as GET, PUT, POST, DELETE.

The client/server mode is not restricted to the nature of the node :

— A device may be a server, and each value measured by a sensor or each action made by a actuator may be modelized through a resource and its associated URI. A client may request a value through a `GET` (for example, `GET /temperature`) or set a state for the actuator (for example, `POST /door "close"`).

— a Device may act as a client and push regularly measured value to a server (for example, `POST /temperature 25`) and periodically check the content of a resource to perform an action (for example, `GET /door`).

Both have pros and cons and can both be implemented simultaneously on the nodes. The former mode allows several applications to access the device's resources, as the latter one imposes the device to send to a specific server. On the other hand, when a device is acting as a client, the application may be informed faster of value changes than when the device is periodically queried.

2.7.1 REST principle

REST principles [4] are widely used, not only for Web sites but also to formalize communication between computers. They establish a strong decoupling between the client and the server, which in traditional computing offers better scalability, and in IoT may preserve energy. The most important concepts are the following :

— A server maintains resources that can be manipulated by clients.

— The resources are identified by a unique and unambiguous identifier, called Universal Resource Identifier (URI).

— The resources have a format from which other URIs can be extracted.

— The server does not maintain any state after processing a request from a client [5]. The state is on the client side.

— The resource may be cached on the client or in intermediary systems.

4. `https://en.wikipedia.org/wiki/REST`
5. we will see later that CoAP with Observe breaks this assumption is some cases.

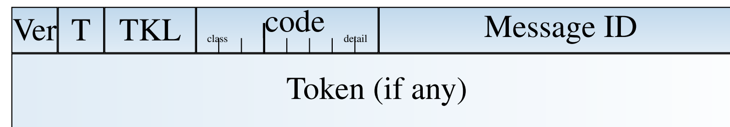


FIGURE 2.5 – Format of a CoAP Header

2.7.2 CoAP Header

Youtube

CoAP is the protocol defined between the client and the server to interact with the resource. CoAP runs on different protocols such as UDP, TCP, SMS, etc. but we will focus on UDP which is the main support. One reason is that UDP is very light and has a limited footprint in both messages and implementation. For SCHC it will be easy to compress, too.

Since UDP, in contrast to TCP, does not provide mechanisms to recover packet losses or to reorder messages, CoAP will implement some lightweight ones.

CoAP keep a structure similar to HTTP, which 3 parts in the message : a mandatory header, followed by options, and, if needed, a payload starting with a separator to distinguish it from the header.

The mandatory header part on 32 bits :

- A 2 bit *version* field indicates the current version 1 as defined in RFC 7252. Therefore, this field always has the binary value `01`, and it will not be the most difficult to compress.
- A *Type* on 2 bits used to make the communication reliable or not :
  - value `00` indicates a confirmable message or `CON` which must be acknowledged by the receiver. `CON`
  - value 01 indicates a non confirmable message or `NON` which does not require any acknowledgment by the receiver. `NON`
  - value 10 indicates an acknowledgment message or `ACK` by the receiver of a `CON` message. This message, as the other, may contain data. `ACK`
  - value 11 indicates a reset message or `RST` in response of a `CON` message when something goes wrong. This message does not contain data. `RST`
- The length of the token on 4 bits, which immediately follows the mandatory header, the authorized values are between 0 (the token does not exist) to 8. Higher values are

reserved.

— A code on 8 bits which cleverly encodes the 3 digit codes used by HTTP. The first 3 bits encode the leftmost digit varying between 1 and 5, the remaining 5 bits encode the 2 remaining digits. For example, the famous 404 error will be encoded as `100_00100`. It could be represented with the hexadecimal value `0x84`, but CoAP prefers the dotted `4.04` to be closer to the HTTP representation. This field is also used to code the requests, setting the first 3 bits to 0 :
  — `0.00` codes empty message, that may be used to an equivalent of ping at the CoAP level.
  — `0.01` for GET that allows a client to access a resource managed by a server.
  — `0.02` for POST which allows a client to create a specific resource on the server.
  — `0.03` for PUT which allows a client to modify the content of an existing resource on the server.
  — `0.04` for DELETE to remove a specific resource from the server.
  — `0.05` for FETCH defined in RFC 8132. In CoAP, a FETCH is equivalent of a GET, but some part of the URI is stored in the payload for compactness, instead of the URI part, which required an ASCII representation.
  — `0.06` for PATCH defined in RFC 8132. It also to modify a part of the structure, in opposition to a PUT which will replace the full structure.
  — `0.07` for iPATCH to signal the server that the request must be idempotent.

— The remaining 16 bits contain the *Message ID*, which will be described in detail in the next sections.

2.7.3 Reliable Messages

Since UDP does not provide any way to recover losses and TCP implies a complex implementation, CoAP offers with the *Types* and the *Message ID* a way to secure transmission. The algorithm is simple. If the sender selects a *Message ID* value and set the type to `CON`, the receiver will acknowledge it with a response containing the same *Message ID* value and a type set to `ACK`.

Of course, nothing is perfect in this world and the `CON` message or the `ACK` can get lost. To cover a lost message, the sender must implement a timer and transmit again the

unacknowledged message with, of course, the same *Message ID*.

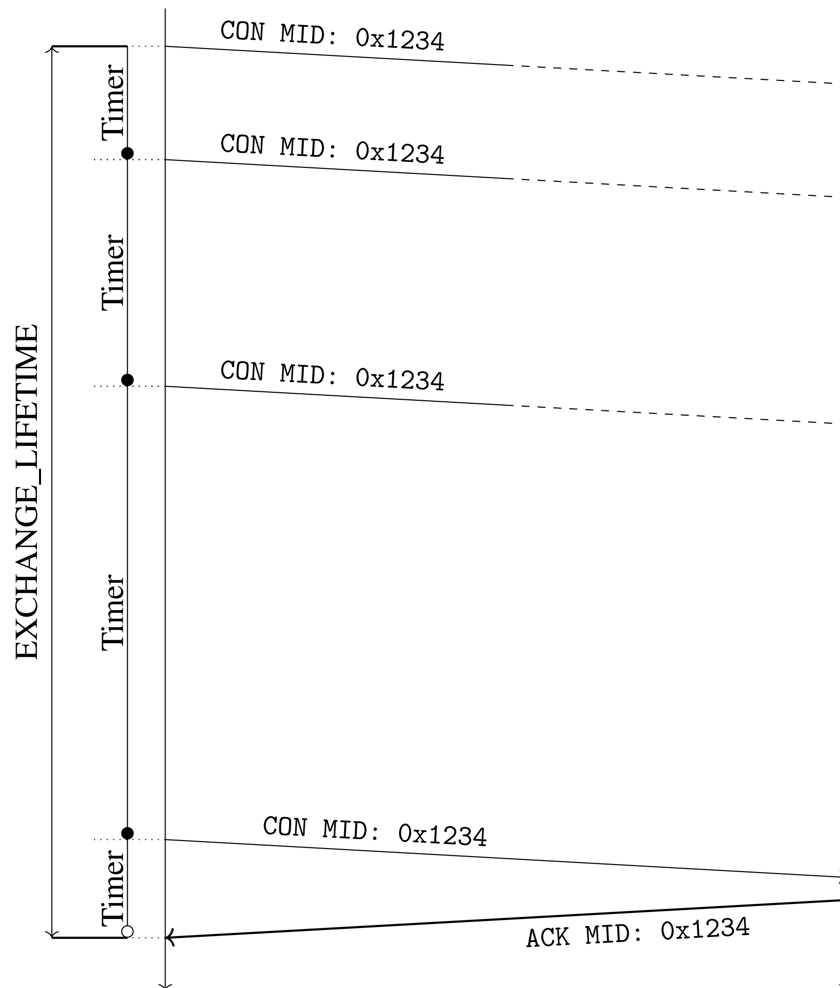


FIGURE 2.6 – Confirmable worst case

It is important to understand how the management of the *Message ID* works in CoAP to optimize the fields for header compression. Figure 2.6 gives an illustration of the worst-case scenario, where all attempts, but the last one, fail. By default, CoAP allows for 4 attempts [6]. A timer triggers retransmission, and the duration doubles at each attempt. To avoid synchronization between nodes, the timer delays are randomized ; the waiting delay can be multiplied by a fact between 1 and 1.5. In the worst-case scenario, with property $\Sigma_{i=0}^{n-1} 2^i = 2^n - 1$, the cumulative timer time is :

$$ACK_TIMEOUT \times (2^{MAX_RETRANSMIT} - 1) \times ACK_RANDOM_TIME$$

corresponding to $2 \times (2^5 - 1) \times 1.5 = 45$ seconds for the maximum delay before successful transmission.

6. In figure 2.6, for readability reasons, only 3 attempts are represented

The computation of maximum delay for a successful transmission implies taking into account the maximum transmission time or latency in both directions and the maximum processing time on the receiver. The CoAP standard assumes that the latency cannot exceed 100s and the maximum processing time is 2s.

Summing all these delays sets the maximum lifetime of a transaction at 247 seconds, which can be approximated to 5 minutes. This is based on default values taken by the standard, but covers most of the common cases.

Of course, there are other scenarios than the one depicted in Figure 2.6 on the facing page. The first transmission may be received, but not the acknowledgments, leading to retransmission of the message. In that case, the receiver will receive multiple copies of the message and will use the Message ID field to recognize them and not process it ; the message will be just acknowledged.

From a compression point of view, if the Message ID fields is reduced to few bits, and the sending frequency is too high, the same message ID will appear again during the 5 minute window and will be considered as a retransmission and not processed.

For non-confirmable messages, it could be considered that the message ID has no importance, since they are not acknowledged, but the underlying layers may duplicate the message ; it can be particularly the case where flooding is used to implement multicast. The Message ID helps the receiver to ignore extra copies. So after receiving a NON message, the receiver will ignore for a period of 145 seconds.

2.7.4 Token

When a client requests a resource, the server may not have immediate access to its content. It can be, for example, a particle sensor that requires one to start a fan and wait a few seconds before returning the value. We saw in the previous section that the retransmission timer is initially set to 2 seconds. So, if the measurement takes more time, the sender will consider that the request was lost and will send it again.

Token

CoAP includes the notion of Token which can be viewed as a kind of connection ; every message that contains the same token will be considered linked.

Figure 2.7 on the next page illustrates the procedure for the use of tokens [7]. The client sends a CoAP request to get access to a measurement. The request is protected with a comfirmable message and includes a token with value `0xFE12`. Since the value acquisition takes time, the server acknowledges the request with an empty `ACK` message with the same Message ID as the request.

When the value is obtained, the server uses a confirmable message to send the response to the client ; the Message ID is generated by the server, but the Token is the same as in

Observe

7. Tokens are also used in combination with the Observe procedure.

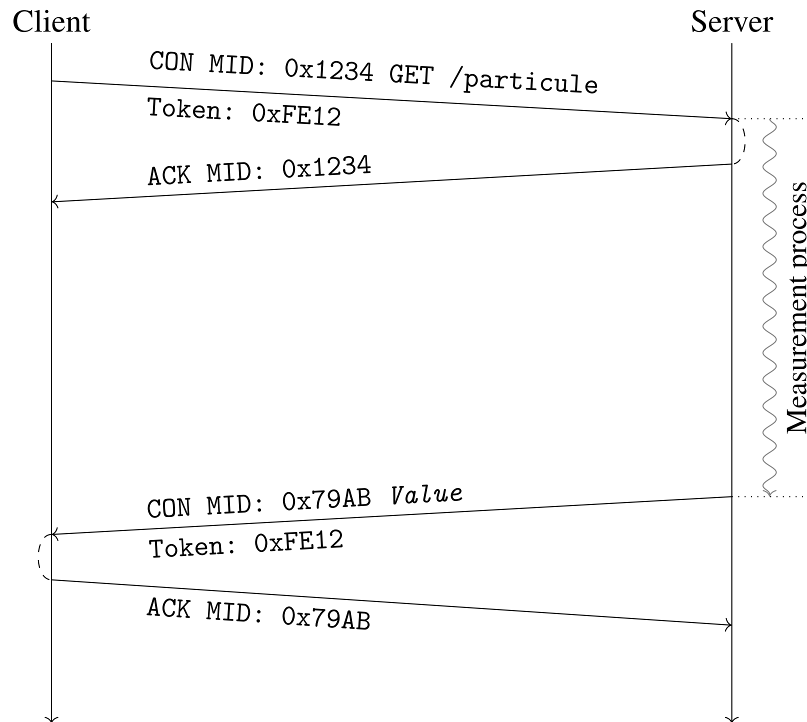


FIGURE 2.7 – Token usage

the original request. The client may establish the relationship between the request and the response.

It could be noted that the confirmable and acknowledgment messages are not related to the REST status. In this example, the client and the server send both of them.

The token is not mandatory; the field `Token Length` tells how many bytes follow the CoAP header to form the token. If not present, `Token Length` is set to 0, any value below 8 indicates the presence of the token.

`Token Length`

In terms of compression, it is always better to avoid the use of tokens if the Message ID can cover the scenario.

Question 2.7.1: CoAP Header

Scapy may not understand CoAP headers and will consider the UDP payload as raw data. Based on `pcap_read.py` write a program that displays the CoAP header and the Token, if exists.

Question 2.7.2: CoAP Port number

Using the previous program, or Wireshark, what is the CoAP Port Number used by the server ?

Question 2.7.3: Message ID

How do the Message ID evolve between two requests ? Is that mandatory ?

2.7.5 Options

Option are essential in CoAP, since the mandatory header does not contain a lot of information, if the extend the CoAP protocol itself, they are called extension, the other carry information to implement the REST paradigm.

Before looking at some CoAP options in more detail, let us see the general way CoAP manipulates options. A unique number is assigned to each option, as shown in Table 2.1 on page 27. Options may transport values ; the value length depends on the option, but may also be different from one message to the other. The table also indicates if an option can be repeated so appearing several times in the CoAP Header.

CoAP protocol optimize the coding of the option into the Header. First, the options are sorted in increasing order, and the delta between two consecutive options is stored on 4 bits. An option starts with the delta if its value is less than 13, otherwise. If the delta is between 14 and 255 + 13, then the value 14 is stored in the 4 bits and the delta minus 13 is stored in the following byte. If the delta is higher than 268, then 15 is stored in the 4 bit field, and two bytes the next two bytes are used to store the real delta.

The 4 bits following the delta are used to store the value length of the option. The same escape encoding is used. if the length is greater than 13, the value is stored in the following bytes. Figure 2.8 on the next page gives the generic option format.

For example, if you have to code two options with values of 5 and 20, the difference is 15. The first option is usually coded with Δ T at 5. For the second option, the ΔT is set to 13 and the next byte will take the value 2.

The way the option are coded are typical form CoAP, during the compression, SCHC will work on an abstract representation of the option.

2.7.6 CoAP Extensions

URI representation

CoAP extensions are quite similar to the options found in HTTP. They are used, for instance, to carry the URI identifying the resource to the server. In HTTP, the URI string has to be parsed to find the different elements. In CoAP, each element is stored in a specific option.

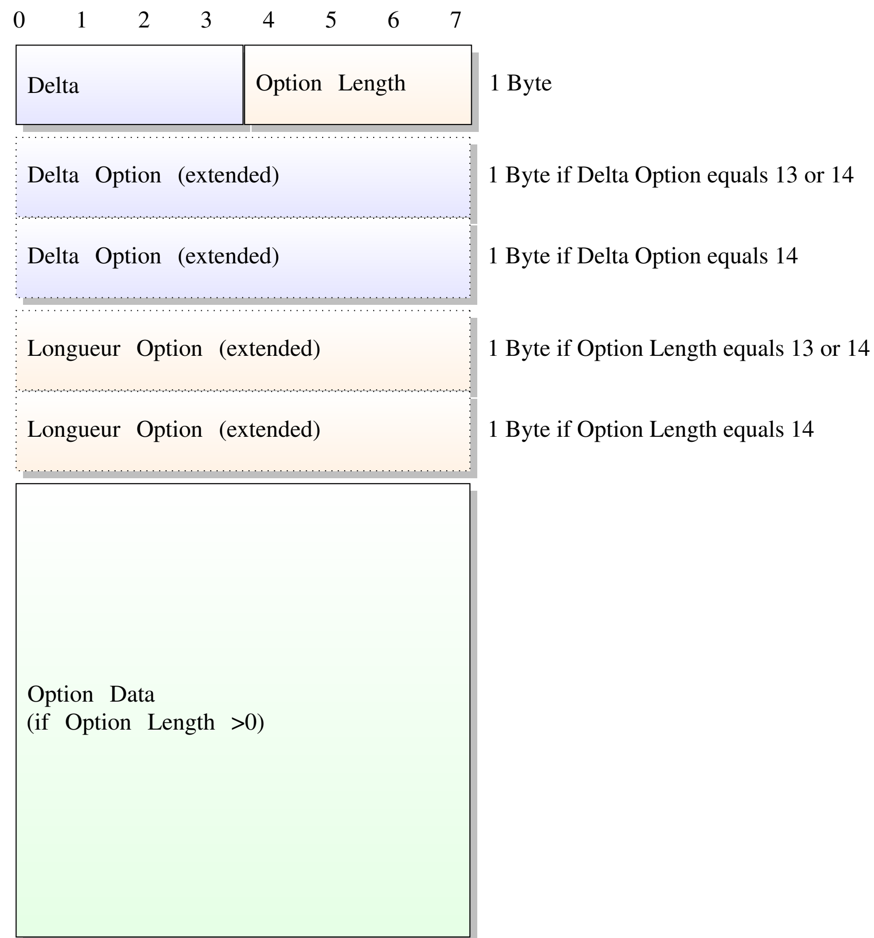


FIGURE 2.8 – Options format

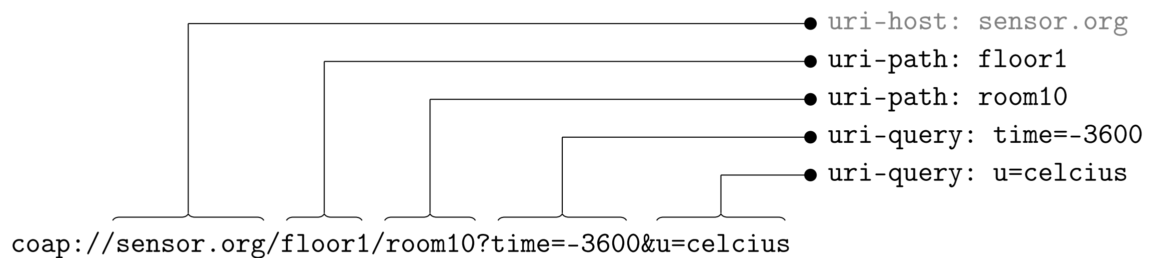


FIGURE 2.9 – URI coding

If-Match
Uri-Host
ETag
If-None-Match
Observe
Uri-Port
Location-Path
Uri-Path
Content-Format
Max-Age
Uri-Query
Accept
Location-Query
Proxy-Uri
Proxy-Scheme
Size1
No-Response

Value	Name	Type	Nature	repeate	Comment
0	reserved				
1	If-Match	opaque	critical	yes	Used to indicate to the server to make the request only under certain conditions.
3	Uri-Host	string	critical		Contains the server name of a URI (name, IPv4 or IPv6 address). Generally, it is not necessary to specify this since CoAP messages are sent to this address.
4	ETag	opaque	optional	yes	Used to manage resource caching
5	If-None-Match	empty	critical		Used to tell a server to make the request only under certain conditions.
6	Observe	integer	optional		Allows a server to send a request to the state changes of a resource. In response, the value must always increase.
7	Uri-Port	integer	critical		Contains the number of the UDP port on which CoAP is launched. Usually, this field is not needed since the CoAP server is already expecting messages on this port.
8	Location-Path	string	optional	yes	Used in response to a POST request to specify a segment of the resource path.
11	Uri-Path	string	critical	yes	Contains one of the segments of the URI
12	Content-Format	integer	optional		Defines the format in which data is encoded
14	Max-Age	integer	optional		The length of time the resource can be cached.
15	Uri-Query	string	critical	yes	
17	Accept	integer	critical		Indicates the formats that the client can accept.
20	Location-Query	string	optional	yes	Used in response to a POST request to indicate the path to the resource.
35	Proxy-Uri	string	critical		Contains a URI that must be taken into account by the proxy.
39	Proxy-Scheme	string	critical		Indicates the encoding scheme.
60	Size1	integer	optional		Indicates the size of the resource.
258	No-Response	integer	optional		Limite les notifications REST

TABLE 2.1 – Some options of the CoAP protocol

Uri-Host
Uri-Path
Uri-Query

Figure 2.9 on the facing page explains how the URI is divided into different elements. The first element `Uri-Host` contains the server name or the IP address. It is immediately followed by `Uri-Path` which are repeated for each element. This simplifies the parsing by the receiver, who can directly access all elements through a repetition of options. If necessary, `Uri-Query` follows and contains each element of the query.

Uri-Host
Uri-Query

When sending the options in a CoAP header, `Uri-Host` may be omitted most of the time, since the server usually runs a single service. Otherwise, this option, as in HTTP, allows one to differentiate which server is called. Table 2.1 indicated that `Uri-Path` option has the value `11`. If this option appears in the first position, the delta will be `11` or `0xb`. For the second instance, the delta will be 0. The option `Uri-Query` has a value 15, so the delta will be `4` for the first instance and 0 for the second.

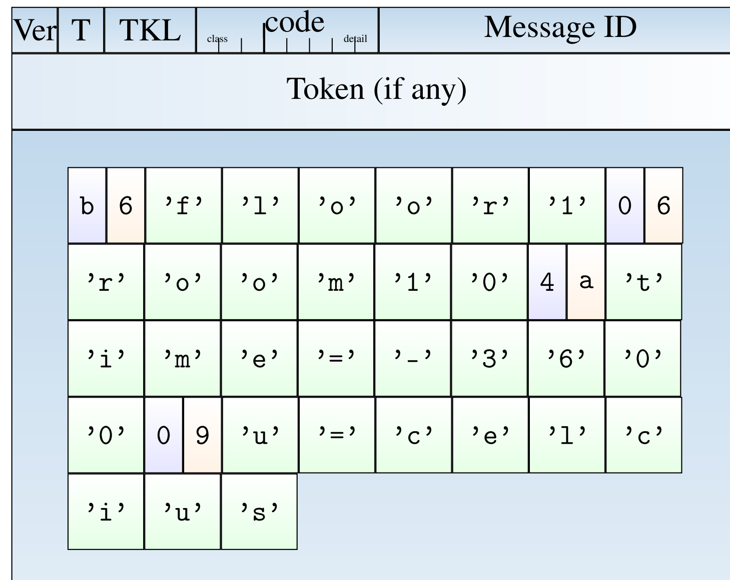


FIGURE 2.10 – Format of a CoAP message

If the request is a GET or a DELETE, the payload is empty, and the CoAP message ends after options. If the request is a POST or a PUT, the payload is usually present. To distinguish between the option and the payload, a special delimiter, called Payload marker, set to value `0xFF`, is inserted between the last option and the payload.

Payload marker

Content formats

Resources information may follow a serialization format such as JSON or CBOR and may also follow certain rules to structure this serialization format with regard to an application.

For example, Table 2.2 on the facing page shows some typical values for the content format. If the value 50 indicates a generic JSON serialization, the value 110 precisions that this JSON format is structured for SENML and the value 11542 for LwM2M.

If no option carries a content format, then value 0 indicating plain text content is assumed.

In a GET request, the `Accept` option indicates to the server which format is expected in the answer. In a response, `Content-Format` is used.

`Accept`
`Content-Format`

Question 2.7.4: Generate a CoAP message

Generate a CoAP Message requesting the content of resource `/sensor/max_temperature` with the CBOR serialization. No token are used.

Value	Type
0	text/plain ; charset=utf-8
40	application/link-format
41	application/xml
42	application/octet-stream
47	application/exi
50	application/json
60	application/cbor
110	application/senml+json
112	application/senml+cbor
11542	application/vnd.oma.lwm2m+tlv
11543	application/vnd.oma.lwm2m+json

TABLE 2.2 – Content formats

Question 2.7.5: Response

What will be the answer if the maximum temperature is 20°Celsius is immediately returned.

No Response

One of the difficulties with CoAP is dealing with two-state machines. `CON` implies an almost immediate response from the receiver acknowledging the message ID. The REST state machine also acknowledges the request by a response. Figure 2.7 on page 24 illustrates the behavior of these two state machines, when the REST response is delayed.

Suppose that we have a particularly reliable network, or that the request contains an update not particularly important. The type `NON` can be used, but it is not possible to suppress the REST response, which will be sent by the receiver to other `NON` requests.

The option `No-Response` is defined in RFC 7967 and allows the client to inform the server which kind of notification can be avoided. Of course, this cannot be used with the method `GET` or `FETCH` that imposes a response with the content. For example, with a `POST`, the client may expect only negative notifications (starting with `4.xx` or `5.xx`) and not positive notifications such as `2.04`.

The `No-Response` option may include a 1 byte bitmap, when the second rightmost bit is set to one, this indicates that the client is not interested by a `2.xx` notification. The same principle applies to the fourth and fifth rightmost bits.

Question 2.7.6: No-Response bitmap

What will be the bitmap value if a client is just interested in `4.xx` and `5.xx` notifications.

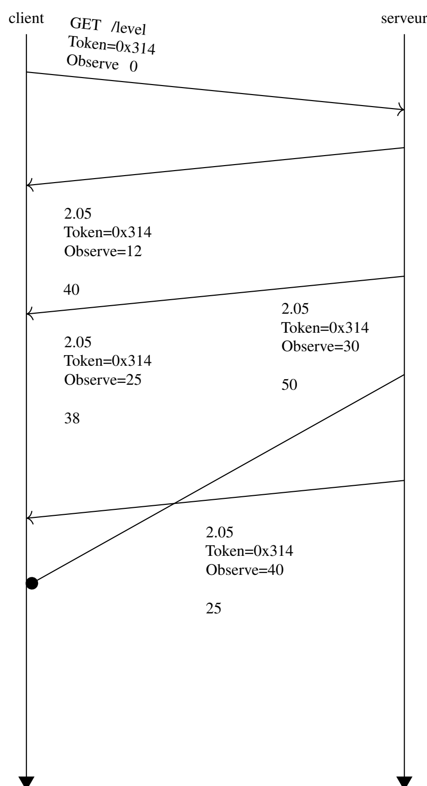


FIGURE 2.11 – Periodic sending with the option `Observe`

Observe

With the REST architecture, the server always responds to a client's request. If we want to follow the evolution of a resource, the client must periodically ask for the value ; the server does not keep any status on past requests. This is not always compatible with the energy constraints of sensors. Suppose that we have a fire alarm that must inform when the smoke level reaches a certain threshold. There are two possibilities :

— The alarm is a REST client and sends a POST to a server when the alert is triggered. For this to work, the alarm must be configured with the address of the server to which to send its POST requests ;

— The alarm is a REST server that has a resource that gives the smoke rate. It does not need to be configured. Clients query it, and it responds to their address. However, if you want to determine when a threshold is reached, you have to continuously query the resource at a high frequency if you don't want to miss any information.

The option Observe, defined in RFC 7641, allows a server to periodically send the value of a resource to the client that requested it. The sending period (or the sending rules such as send when a threshold is reached) is defined by the behavior of the server.

Observe

Figure 2.11 on the facing page illustrates this behavior. The client sends a GET request with the Observe option set to 0, but in value. If the client accepts this option, it will respond by setting it in its responses. In this case, it must have a value that can only increase from response to response. This is useful to allow the client to detect a desequencing of the responses. Thus in the example, the Observe stamp 30 arrives after the one stamped 40 and will be rejected by the client.

Token

Note also that the Token field must be present to make the link between the request and the responses.

It must also be possible to stop an Observe. There are several cases :

— The client wants to stop an Observe in progress. He redoes the same request but sets the Observe option to 1.

— the client restarts, it may lose its context about the Observe, but continue to receive periodic requests from the server. The client not recognizing the Token, sends a message ReSeT. The server cancels the periodic transmission to the client.

ReSeT

— the client is unreachable, it will not be able to cancel the transmission. As a rule, responses with the Observe option are carried in confirmable CON messages. The server may occasionally send the response in a confirmable CON message. If it does not receive an acknowledgment from the client, it deduces that it has disappeared and stops sending periodic responses.

CON

CON

2.8 Hand-on CoAP

aoicoap

In this section, we are going to study how CoAP header are formed using the `aoicoap`[8] Python implementation.

The package includes two programs : `aiocoap_fileserver` is a CoAP server that allows file contents to be retrieved in a directory, and `aoicoap_client` which helps create CoAP requests. For the purpose of this hands-on, both programs will communicate in IPv6 through the loopback interface and the associate `::1` IPv6 address[9]. On the same computer, we will start a Wireshark[10] protocol analyzer, listen to the loopback interface, and filter traffic for UDP on the default allocated port 5683[11].

The first step is to start the CoAP server in a terminal window. Type :

Listing 2.2 – CoAP Server

```
%aiocoap-fileserver --bind '[::1]:5683' /bin
```

8. Type `pip3 install aiocoap`

9. to avoid confusion in the notation with a port number, the IPv6 address is surrounded with square brackets which must be themselves quoted to not be interpreted by the Linux shell.

10. see `wireshark.org` for installation.

11. When the capture is started, type in the filter window `udp.port==5683`

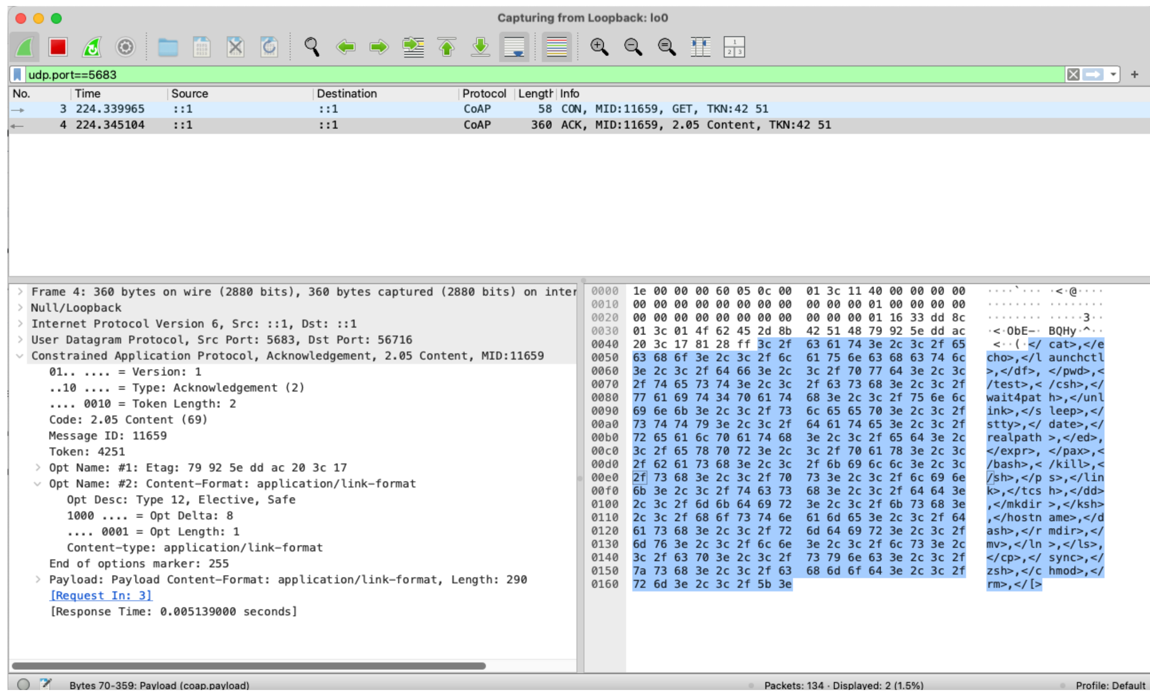


FIGURE 2.12 – CoAP exchange captured by Wireshark.

This starts the server on the loopback address and the default port. The fileserver sees `/bin` as the top directory [12].

On another terminal, send a CoAP request through the Command Line Interface with wireshark active.

Listing 2.3 – CoAP client

```
%aiocoap-client -v "coap://[::1]"
</cat>,</echo>,</launchctl>,</df>,</pwd>,</test>,</csh>,</wait4path>,
</unlink>,</sleep>,</stty>,</date>,</realpath>,</ed>,</expr>,</pax>,
</bash>,</kill>,</sh>,</ps>,</link>,</tcsh>,</dd>,</mkdir>,</ksh>,
</hostname>,</dash>,</rmdir>,</mv>,</ln>,</ls>,</cp>,</sync>,</zsh>,
</chmod>,</rm>,</[>
```

This type of network traffic is visible in the Wireshark interface on **??** on page ??.

Question 2.8.1: URI

What is the URI in the previous command ?

Analyze the traffic on Wireshark.

12. IPv6 addresses use square brackets to distinguish them from port numbers in URL syntax.

Question 2.8.2: No options

Why the request contains no option ?

Question 2.8.3: Response

What are the options in the response ?

Question 2.8.4: Bad request

Forge a request, for example `coap://[::1]/foo/bar?q=1r=0`, and ask for a content in CBOR (value 60). What are the CoAP options ?

Question 2.8.5: Client

What is the goal of the `-non` option in the `aiocoap-client` program ?

3. SCHC architecture elements

The compression-decompression and fragmentation-reassembly [1] mechanisms of SCHC are based on a point-to-point association and a common static context shared by both of these ends (cf. Figure 3.1 on the facing page). This architectural concept differs from traditional networks, where a protocol is developed to cover a large number of situations. With SCHC, the behavior may be adapted to a specific case. Of course, this is an architectural model and in practice, the static contexts will be defined for classes of traffic pattern and usage.

A SCHC instance is defined as two End-Point sharing the same static context. The context is defined as static because it does not evolve directly with the sending or receiving of SCHC packets. The context contains a Set of Rules (abbreviated in SoR) dedicated to compression or fragmentation [2].

SCHC instance

Set of Rules

SCHC has been originally designed for constrained devices and networks, so one of these end points represents a device. The device traffic issue is designed as uplink traffic [3].

1. To be more, we can also refer to C/D or F/R mechanisms, but most of the time when we talk of compression rules, we include the decompression process.
2. The way the context is provisioned in both End-Point is out of the scope of the compression and fragmentation process and being seen latter.
3. In the rest of this book, we will try to represent the device in red color and at the left hand-side of the figure.

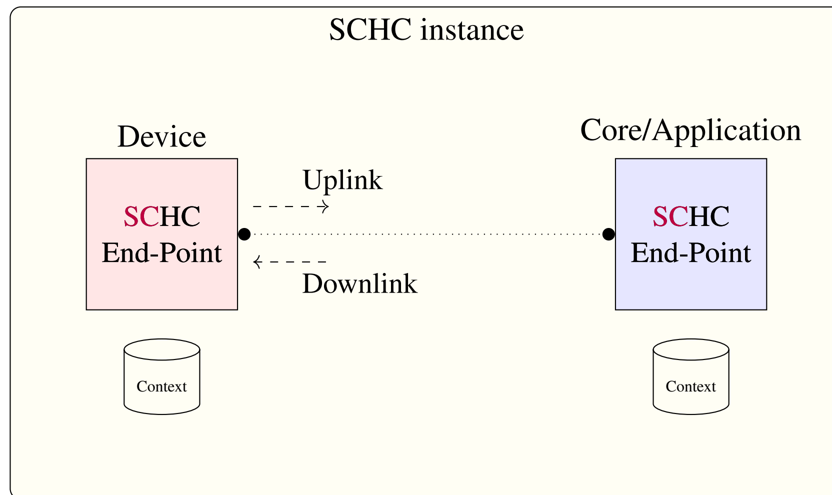


FIGURE 3.1 – SCHC Architecture.

In contrast, the traffic toward the device is seen as the downlink traffic. The other one is implemented in a core equipment acting as a router or directly on application, depending on the use case.

The device usually contains a small set of compression or fragmentation rules that are tailored to the traffic generated and received by the device only. Conversely, the core manages several instances of SCHC and must know all the devices' rule sets. Even if all devices are identical, the SCHC core treats each device as if it were unique.

3.1 Rule ID's

rule ID

Rules in a static context are identified by rule ID. A rule ID is a binary sequence that must be unique within the instance. RFC 8724 does not specify the length of the rule ID's : the lengths are chosen when the rules are defined, and the lengths can even be different for different rules within the same set. Within a set of rules, the rule ID representations must not overlap when read from left to right.

For example :

— `0` and `111111` are two valid rule ID's

— `01` and `0101` is not a valid pair of rule ID's within the same set : on receiving the sequence of bits 0101 (from left to right), it is impossible to tell which rule is being received.

Rule IDs could be described using their binary representation, but for compactness, the decimal notation `rule ID value/rule ID length` will be used. For example, `3/8` indicates a rule ID represented in 8 bits and having a decimal value of 3 (right aligned), which is equivalent to the `00000011` binary representation.

If the slash notation is compact and very close to the prefix notation in IP, take note that this notation can be misleading. For example,

— `1/2` and `5/4` don't look overlapping, even though they do (binary `01` and `0101`);

— `1/2` and `5/4` don't look like they overlap, even though they do (binary `01` and `0101`); as another example, the pair `12/4` and `12/6` looks suspicious even though it is perfectly valid (binary `1100` and `001100`).

.

Figure 3.2 gives an example of the binary tree representation. Some leaves are defined for compression, others for fragmentation. Compression rules are usually bidirectional; the same rule ID can be used in uplink and downlink traffic. In contrast, the fragmentation is unidirectional; to allow fragmentation in both directions, at least two rules must be defined. This choice is driven by the desire to allow different fragmentation behavior in each direction since the characteristics may differ.

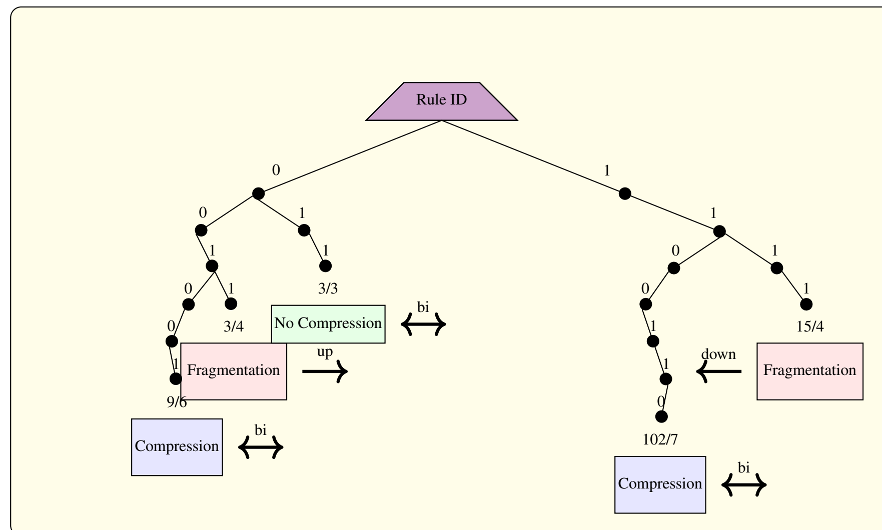


FIGURE 3.2 – Example of binary tree associated to rule IDs

Figure 7.1 on page 101 illustrates the combination of compression and fragmentation rules. An uncompressed packet arrives, the compressor selects rule 9/6 but remains to be sent on one piece. The compressor selects rule 3/4 to send the fragments. The receiver checks the integrity of the message and may return an acknowledgment to the fragmenter and send the reassembled message to the decompressor which generates the original packet.

The application that receives the responses to the packet and the same rule 9/6 is used to compress the answer [4]. Fragmentation is done using this time rule 15/4. Since some fragments are lost the receiver asks for their retransmission, then checks integrity, acknowledges the fragmenter, and sends the reassembled message to the decompressor.

We see that since a direction is assigned to the fragmentation rules, the other direction is used to carry protocols information such as acknowledgment or an indication of the missing fragments.

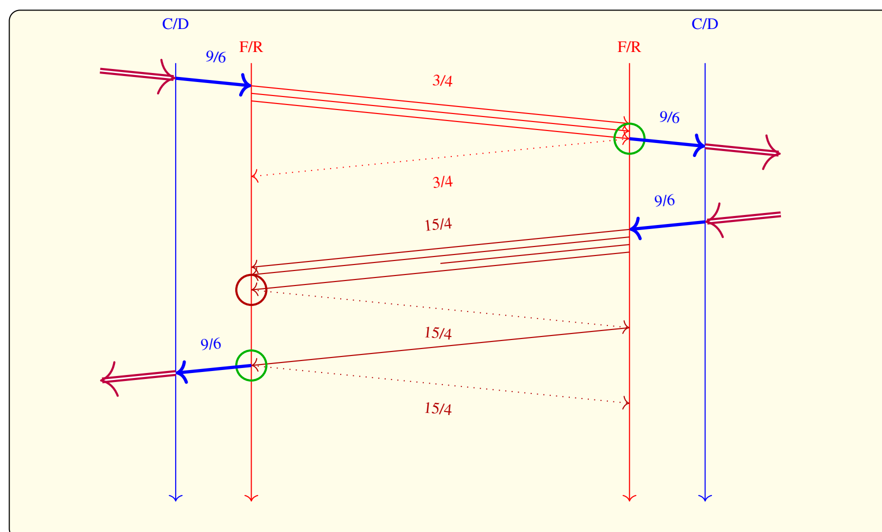


FIGURE 3.3 – Compression and Fragmentation combined

3.2 Rules structure

Each SCHC implementation has its own rule format adapted to the programming language used and the scope of the solution. In this section, we are going to focus on the OpenSCHC representation which is based on the JSON format.

A Rule is a JSON Object with three keys :

RuleID
— the key `RuleID` gives the rule ID value,

RuleIDLength
— the key `RuleIDLength` indicates on how many bits the ruleID is coded,

4. This is not mandatory, another rule may have been selected.

— the last key gives the nature of the rule. The key can be either :

  — `Compression`, the value is a JSON array containing the packet fields.

  — `Fragmentation`, the value is a JSON Object containing the fragmentation parameter,

  — `NoCompression`, the value is null or an empty array.

Compression

Fragmentation

NoCompression

For example, a compression rule

```
{
 "RuleID" : 38,
 "RuleIDLength" : 8,
 "Compression": [
   {"FID": "IPV6.VER", "TV": 6, "MO": "equal", "CDA": "not-sent"},
   {"FID": "IPV6.TC",  "TV": 0, "MO": "ignore", "CDA": "not-sent"},
   {"FID": "IPV6.FL",  "TV": 0, "MO": "ignore","CDA": "not-sent"},
   ...
```

```
    {
    "RuleID" : 12,
    "RuleIDLength" : 6,
    "Fragmentation" : {
      "FRMode" : "NoAck",
      "FRDirection" : "UP"
    }
```

Finally, the `NoCompression` keyword is used to describe the mandatory default rule that SCHC resorts to for sending a packet when no valid compression rule was found to compress it.

NoCompression

```
   {
   "RuleID" : 666,
   "RuleIDLength" : 10,
   "NoCompression" : []
   }
```

The fragmentation, compression, and no-compression rules share the same rule ID space : the same rule ID cannot be used for both a compression rule and a fragmentation rule, for example.

3.3 The Rule Manager

The Rule Manager plays an important role in the openSCHC implementation of SCHC. It has multiple goals :

Rule Manager

— check for the correctness of the rule's external description,

— add default parameter values, which allows simplifying the rules external description,

— import external rule descriptions into internal storage,

— display the rules in the store in a format more compact than JSON,

— find in the store a valid rule to compress or fragment an incoming packet with,

— retrieve a rule from the store by its rule ID.

3.3.1 Adding Rules into the Rule Manager

Listing 3.1 shows a simple Python program used to manage the two rules described in Section 3.2.

Listing 3.1 – rm1.py

```python
from gen_rulemanager import *

RM = RuleManager()

rule1100 =   {
  "RuleID" : 12,
  "RuleIDLength" : 4,
  "Compression" : []
}

rule001100 =   {
  "RuleID" : 12,
  "RuleIDLength" : 6,
  "Fragmentation" :  {
    "FRMode" : "noAck",
    "FRDirection" : "UP"
  }
}

RM.Add(dev_info=rule1100)
RM.Add(dev_info=rule001100)

RM.Print()
```

RuleManager

It first imports the `gen_rulemanager` module, which defines the `RuleManager` class.

gen_rulemanager

Add

The `Add` method adds a new rule to the store.

gen_rulemanager

Print

The `Print` method displays the following :

gen_rulemanager

```
****************************************
Device: None
/-------------------------\
|Rule 12/4           1100  |
|---------------+---+--+--+------------------------------+-------------+----------------\
\---------------+---+--+--+------------------------------+-------------+----------------/
/-------------------------\
|Rule 12/6         001100  |
!=========================+==============================================================\
!^ Fragmentation mode : noAck    header dtag 2 Window  0 FCN  3                       UP ^!
!^ No Tile size specified                                                              ^!
!^ RCS Algorithm: crc32                                                                ^!
\========================================================================================/
```

Compression rules normally contain header field descriptors (omitted in this simple example), and fragmentation rules contain the fragmentation parameters. Note that for the

fragmentation rule used here, the Rule Manager added some default parameters correspon-ding to the No-Ack behavior of SCHC. No-Ack

3.3.2 Adding a Set of Rules

A device should contain a set of rules related to compression and fragmentation. In openSCHC, the SoR (Set of Rules) is a JSON array. The following program has the same behavior as the one shown in 3.3.1, but the rules are added with a single method call, using an array. Set of Rules

Listing 3.2 – rm2.py

```
from gen_rulemanager import *

RM = RuleManager()

rule1100 =   {
 "RuleID" : 12,
 "RuleIDLength" : 4,
 "Compression" : []
}

rule001100 =   {
 "RuleID" : 12,
 "RuleIDLength" : 6,
 "Fragmentation" : {
   "FRMode": "noAck",
   "FRDirection" : "UP"
 }
}

RM.Add(dev_info=[rule1100, rule001100])

RM.Print()
```

3.3.3 Attributing a Set of Rules to a Device

You may have noticed that, in the previous examples, the device was displayed as None. This is appropriate when SCHC is instantiated on a device, since there is no ambiguity as to which device the rule set applies to. In contrast, when the SCHC instance resides on the core network side, each set of rules must be associated with a distinct device.

In openSCHC, the DeviceID is structured as a link technology and an identifier within that link technology : DeviceID

— if the link technology is a UDP tunnel (as in the test platform described in Chapter **??** on page ??), the technology keyword is `udp` and the identifier is the device-side tunnel endpoint IP address, followed by the port number used on that endpoint. Therefore, a full DeviceID would be `udp:83.199.24.39:8888`[5]. UDP tunnel

5. If the device is behind a NAT, the IP address used must be the global address assigned to the NAT.

LoRaWAN

— for LoRaWAN (*not yet implemented*), the technology keyword is `lorawan` and the identifier is the *devEUI*.

Add

gen_rulemanager

While being stored in the rule manager via the `Add` method, rules can be attributed to a DeviceID as follows :

```
RM.Add(device="udp:83.199.24.39:8888", dev_info=[rule1100, rule001100])
```

Alternately, the following JSON structure could be used :

```
{
    "DeviceID": "udp:83.199.24.39:8888",
    "SoR" : [ ..... ]
}
```

3.3.4 Importing rules from a JSON file

Rules can also be described in a JSON file, in which case the Rule Manager`Add` method is used with the `file` argument. For example :

```
rm.Add(file="icmp.json")
```

This approach allows for easier management and sharing of rule sets, as they can be stored in separate JSON files and loaded as needed. It's particularly useful when dealing with complex or numerous rule sets, as it keeps the main code clean and allows for easier version control of the rules. The JSON file would contain the rule definitions in the same format as we've seen earlier, possibly including multiple rules for both compression and fragmentation. This method of importing rules provides several advantages :

— Separation of concerns : The rules can be managed separately from the code that uses them.

— Flexibility : Rules can be easily updated or swapped out by changing the JSON file, without needing to modify the Python code.

— Readability : JSON is a human-readable format, making it easier to review and edit rule sets.

— Portability : JSON files can be easily shared between different systems or implementations.

When using this method, it's important to ensure that the JSON file is well-formed and follows the expected structure for SCHC rules. The Rule Manager will typically perform some validation when importing the rules, but it's a good practice to verify the JSON file's correctness before attempting to import it.

It's important to note that the JSON format described here is specific to the openSCHC implementation. Other SCHC implementations may use different formats or data structures to represent and manage rules. To ensure interoperability between different SCHC implementations, it is necessary to use a standardized format. The IETF has defined such a format in RFC 9363, which specifies a YANG Data Model for SCHC. This YANG model provides

a vendor-neutral, implementation-independent way to describe SCHC rules and contexts. When working across different SCHC implementations or when aiming for broader interoperability, it's recommended to use the YANG Data Model defined in RFC 9363 rather than implementation-specific formats like the JSON structure used in openSCHC.

3.4 Exercises

Question 3.4.1: Dual rules

It is possible to have the same RuleID for a compression rule and a fragmentation rule.

- True
- False

Question 3.4.2: RuleID in binary

Give the binary representation of Rule ID `12/5`

Question 3.4.3: binary tree

Which rules are compatible with `12/5` ?

- `3/3`
- `13/5`
- `24/6`
- `26/6`
- `12/6`

4. Compression

If you're looking to gain a deeper understanding of SCHC header compression and its distinctive approach to managing header fields, this chapter will provide valuable insights.

This chapter delves into the topic of header field compression in SCHC and the differences it presents compared to standard packet dumps.

4.1 Compression rule

SCHC compression is not limited to a specific protocol. SCHC does not establish a context during transmission and between both ends. The compression takes the generic notion of fields composing a header in order. Therefore, a compression rule identifies which field the compression can apply without ambiguities and which kind of encoding SCHC uses. SCHC compresses datagrams and does not have the notion of flows. SCHC defines the description of the header information with the knowledge obtained after analyzing the header field values.

A compression rule contains descriptors to define header information in several lines representing the header expected in the packet. If the fields in the packet differ from the ones in the rule, the rule must not apply, and SCHC must look for another rule. A field header description is in a line of the rule, which has seven descriptors, regrouped in three parts :

Each line is divided into 3 parts :

— the field identification

— the comparison method

— if the rule is selected, how the field compression occurs.

4.1.1 Field identification.

Field identification is composed of several elements. The first one is a field ID. The RFC 8724 do not impose each implementation on having the same way to name the fields. Table 11.2 on page 146 gives some Field IDs for IPv6 and UDP headers. Note that to allow exchanges of rules, RFC 9363 defines, with a YANG Data Model, some unique identifiers for these field IDs.

RFC 8724	openSCHC	RFC 9363 (YANG DM)	Length (bits)	Position
IPv6 Version	IPV6.VER	fid-ipv6-version	4	1
IPv6 Diffserv	IPV6.TC	fid-ipv6-trafficclass	8	1
IPv6 Flow Label	IPV6.FL	fid-ipv6-flowlabel	20	1
IPv6 Length	IPV6.LEN	fid-ipv6-payload-length	16	1
IPv6 Next Header	IPV6.NXT	fid-ipv6-nextheader	8	1
IPv6 Hop Limit	IPV6.HOP_LMT	fid-ipv6-hoplimit	8	1
IPv6 DevPrefix	IPV6.DEV_PREFIX	fid-ipv6-devprefix	64	1
IPv6 DevIID	IPV6.DEV_IID	fid-ipv6-deviid	64	1
IPv6 AppPrefix	IPV6.APP_PREFIX	fid-ipv6-appprefix	64	1
IPv6 AppIID	IPV6.APP_IID	fid-ipv6-appiid	64	1
UDP DevPort	UDP.DEV_PORT	fid-udp-dev-port	8	1
UDP AppPort	UDP.APP_PORT	fid-udp-app-port	8	1
UDP Length	UDP.LEN	fid-udp-length	8	1
UDP checksum	UDP.CKSUM	fid-udp-checksum	8	1

TABLE 4.1 – Field ID definition in different implementations

For IPv6 headers, the version, traffic class, flow label, upper layer protocol, hop limit, and payload length are the same as in a standard packet dump. However, when it comes to addresses, SCHC treats device and application addresses separately and does not follow the fields in the packet. As a result, the addresses remain constant regardless of who emitted the

packet. Additionally, SCHC breaks down addresses into two parts : the prefix on 64 bits and the Interface Identifier (IID) on 64 bits, which streamlines compression and eliminates the need to write two rules for device and application packets.

This same approach applies to UDP port numbers, which SCHC assigns to the device and application. The UDP header then continues with the length and checksum fields.

Field Length and Field Position.

Field Length

Field Length, Field Position and Direction Indicator are not so important for IPv6 and UDP since they have a very strict format, but may be useful for other protocols such as CoAP.

As introduced in table 11.2 on page 146, the Field Length indicates, in bits, the length of the field. Since IPv6 and UDP have fixed length fields this information is not in that case that much important/ But, the Field Length could be used to modify the prefix length, if the prefix length is different of 64.

Field Position

In the same table, all the fields are tagged with 1 int the Field Position column. This is also a specific case for the IPv6 and UDP protocol, where a field appears only once in a header.

Direction Indicator.

Direction Indicator

Direction Indicator tells to take account of the field regarding its direction. As we saw for the IPv6 addresses and the UDP port numbers, one host is identified as the Device, and the other as the Application. Communication between the Device and the Application is in the up-link direction and down-link in the reverse direction.

up-link
down-link

A field is tagged as :

bi-directional

- bi-directional, if it appears in both directions.
- up-link, if it appears only in traffic from the Device to the Application.
- down-link, if it appears only in traffic from the Application to the Device.

For IPv6 and UDP, the all field can be tagged bi-directional, since they appears in all exchanges. The use of device and application addresses instead of source and destination addresses at the IP level and device and application ports at the UDP level, makes the rule insensible to direction. Nevertheless, the Hop Limit field may benefit of splitting the traffic in two directions. Traffic coming from the device does not cross any routers. In the opposite direction, the number of routers will vary over time.


```
0000  60 0A 45 F8 00 1F 11 40 20 01 41 D0 03 02 22 00  ‘.E....@ .A...".
0010  00 00 00 00 00 00 13 B3 20 01 41 D0 04 04 02 00  ........ .A.....
0020  00 00 00 00 00 00 3A 86 16 33 81 B9 00 1F 3D 12  ......:..3....=.
0030  62 45 9E F8 3E C5 FF 32 30 32 33 2D 30 34 2D 30  bE..>..2023-04-0
0040  36 20 31 30 3A 31 30                             6 10:10
```

```
###[ IPv6 ]###
     version   = 6
     tc        = 0
     fl        = 673272
     plen      = 31
     nh        = UDP
     hlim      = 64
     src       = 2001:41d0:302:2200::13b3
     dst       = 2001:41d0:404:200::3a86
###[ UDP ]###
        sport     = 5683
        dport     = 33209
        len       = 31
        chksum    = 0x3d12
###[ Raw ]###
           load      = 'bE\\x9e\\xf8>\\xc5\\xff2023-04-06 10:10'
```

Question 4.1.1: IPv6 fields identification.

Considering the packet above, in hexadecimal and disassembled.

What is the value of the `IPV6.VER` field ?

— 4
— 40
— 6
— 60

The direction is down-link, what is the value of the `IPV6.DEV_PREFIX` field ?

— 20 01 41 D0 03 02 22 00
— 00 00 00 00 00 00 13 B3
— 20 01 41 D0 04 04 02 00
— 00 00 00 00 00 00 3A 86

4.1.2 Field Matching.

Youtube

The field matching is based on two criteria, the Target Value (TV) containing the expected value for that field in the packet, and the Matching Operator (MO) indicating how the comparison should be done.

The RFC 8724 defines 4 matching operators :

— Ignore : there is no comparison between the Target Value and the value for that field Ignore

in the packet.

Equal
— Equal : the value in the Target Value must be the same as the one found in that field in the packet.

MSB
— Most Significant Bit (MSB) : This MO takes an argument indicating on how many bits the comparison between the TV and the value on the field must be done.

Match-Mapping
— Match-Mapping : The Target Value contains a list of values, and one of these values must be equal to the value for that field.

Target Value
openSCHC
The Target Value by itself is either a non-typed byte string or, for Match-Mapping, an array of non-typed byte strings. For easier readability, different types may be used. For example, for better readability, openSCHC allows the use of strings, integers, and even IPv6 addresses.

Question 4.1.2: MSB

What are the bits tested for a MSB of 12 bits and a Target Value of 23628.

— `111000100110`

— `101110001001`

— `100110100100`

— `010001011100`

Question 4.1.3: Matching List

The traffic issued from the device and the application may both use IPv6 global and link-local addresses. What will be the Target Value to describe these device prefixes ?

A rule can be selected if and only if all the fields in the rule match all the fields in the packet header. If one Matching Operator fails or if the packet contains some extra fields, some Field Descriptors in the rule will not be found in the packet, and the rule cannot be selected.

4.1.3 Compression

If a rule is selected through the matching process described above, compression can occur. The Compression Decompression Action (CDA) describes how the compression is performed, forming the residue that directly follows the ID of the rule. The decompression will do the opposite operation taking the residue and combining it with the Target Value.

The RFC 8724 proposes 7 actions :

— not-sent : the value in the field is elided and will be found in the Target Value. not-sent

— value-sent : the value is the field is sent as part of the residue. value-sent

— Less Significant Bit (LSB) : the Less Significant Bit which where not part of the MSB comparison are sent. This CDA is only valuable with the MO MSB.

— mapping-sent : the index 0 in the Target Value where the value of the Field has been found. The size of the index is the smallest with respect to the number of entries in the Matching List. For instance, if the Matching List contains 5 entries, the residue will be 3 bit long. mapping-sent

— compute-length : The value is not sent, and the receiver will compute it from the data it receives after decompression. compute-length

— compute-checksum : The value is not sent, and the receiver will compute it from the data it receives after decompression [1]. compute-checksum

— devIID : The device Interface Identifier is derived from the layer 2 device address. devIID

— appIID : The application Interface Identifier is derived form the layer 2 application address [2]. appIID

4.1.4 Decompression

Decompression uses the same CDA which tells how to combine the Target Value and the residue. When a SCHC packet arrives :

1. the decompressor identifies the sender and its associated Set of Rules,
2. since the ruleID is organized as a binary tree, the RuleID value and length is easily extracted.
3. the rule is identified in the Set or Rules which specify the residue format.

4.2 Simple compression rules for openSCHC.

In the openSCHC implementation, a JSON object describes, as an Array, the set of rules associated with a device. Each element is an object defining the rule. A rule must contain three keywords :

1. As you notice the description is identical to compute-length, RFC 9363 introduces the concept of compute-* to regroup these two CDAs.

2. This CDA is less applicable than devIID CDA since the application is usually not located in the core SCHC and therefore not present in Layer 2 frames.

RuleIDValue
RuleIDLength

— RuleIDValue and RuleIDLength. As presented before, a rule ID is composed of two elements, the Value and the length in bits to store with value.

— the nature of the rule, though a specific and exclusive key : Compression, No-Compression or Fragmentation.

The example, listing 4.1, gives an overview of the structure with an empty compression rule.

Listing 4.1 – Empty compression rule

```
[
  {
  "RuleIDValue" : 12,
  "RuleIDLength" : 4,
  "Compression" : []
  }
]
```

Compression

The listing 4.2 on the next page gives an example of an openSCHC IPv6 header compression rule written in JSON. The array defined by the Compression key contains all the Field Descriptors. They are Objects with specific keywords. For readability, openSCHC does not require all the elements in the Field Descriptors. In our example, the Field Length, the Field Position and the Field Direction are not present ; openSCHC will fill them with default values.

The keys used by openSCHC for Field Descriptors are [3] :

— Field IDentifier (FID). This mandatory key indicates the Field ID. Table 11.2 on page 146 gives an overview of some identifiers.

— Field Length (FL) indicates in bits the length of the field. It can be either a positive integer for fixed length fields or a function for variable length fields (see XXX).

— Direction Indicator (DI) indicates the expected direction of the packet ; it can be the string UP, DW or BI. By default, a field is considered bidirectional.

— TV indicates the expected value found in the packet. openSCHC allows different formats, such as integer, strings, or IPv6 addresses and prefixes as shown line 24. The TV can also be an array of theses values [4].

— MO defines the Matching Operator. OpenSCHC implements MO defined in RFC 8724 : EQUAL, IGNORE, MSB, MATCH-MAPPING.

MO.VAL

— MO.VAL for MO MSB, this entry specifies how many bits the comparison will take.

— CDA defines the compression or decompression algorithm. openSCHC implements : NOT-SENT, VALUE-SENT, LSB, MPPING-SENT, COMPUTE-LENGTH, COMPUTE-CHECKSUM [5].

3. the keys are stored in the gen_parameters.py file.
gen_parameters.py
4. The value null extends the array length and therefore the residue size.
5. DEVIID and APPIID are not yet implemented.

Listing 4.2 – IPv6.json

```
[{
    "RuleIDValue" : 5,
    "RuleIDLength": 3,
    "Compression": [
      {"FID": "IPV6.VER",
       "TV": 6, "MO": "equal",
       "CDA": "not-sent"},
      {"FID": "IPV6.TC",
       "TV": 1, "MO": "equal",
       "CDA": "not-sent"},
      {"FID": "IPV6.FL",
       "TV": 144470, "MO": "equal",
       "CDA": "not-sent"},
      {"FID": "IPV6.LEN",
       "MO": "ignore",
       "CDA": "compute-length"},
      {"FID": "IPV6.NXT",
       "TV": 17, "MO": "equal",
       "CDA": "not-sent"},
      {"FID": "IPV6.HOP_LMT",
       "MO": "ignore",
       "CDA": "value-sent"},
      {"FID": "IPV6.DEV_PREFIX",
       "TV": ["FE80::/64",
             "2001:41D0:302:2200::/64"], "MO": "match-mapping",
       "CDA": "mapping-sent"},
      {"FID": "IPV6.DEV_IID",
       "TV": "::13b3", "MO": "equal",
       "CDA": "not-sent"},
      {"FID": "IPV6.APP_PREFIX",
       "TV": ["2001:41d0:404:200::/64",
              "FE80::/64"], "MO": "match-mapping",
       "CDA": "mapping-sent"},
      {"FID": "IPV6.APP_IID",
       "TV": 2,"MO": "equal",
       "CDA": "not-sent"},
      {"FID": "UDP.DEV_PORT",
       "MO": "ignore",
       "CDA": "value-sent"},
      {"FID": "UDP.APP_PORT",
       "TV": 5680,"MO": "MSB",
       "MO.VAL": 12, "CDA" : "LSB"},
      {"FID": "UDP.LEN",
       "TV": 0,
       "MO": "ignore",
       "CDA": "compute-length"},
      {"FID": "UDP.CKSUM",
       "TV": 0, "MO": "ignore",
       "CDA": "compute-checksum"}
      ]
```

```
    }
52 ]
```

Question 4.2.1: Residue Length

Considering the rule given listing 4.2 on the preceding page. What is the length of the residue ? What is the size of the SCHC packet (assuming that the data part is 40 byte long) ?

Question 4.2.2: Rule

Does the following rule match the packet given page 45 matches the rule given in listing 4.2 on the facing page.

Question 4.2.3: SCHC packet

Decompress the following up-link SCHC packet `a4 6e b1 7a a4 34 90 86 85 00` [a].

a. To help you, this is the binary equivalent : `101 00100011 0 1 1101011000101111 0101 01001000 01101001 00100001 00001101 00001010 0000000`

4.3 Hands-on

4.3.1 Parsing

Youtube

pcap

The Python program `pcap_parse.py` displays the content of a pcap file that contains packets captured on a network. Let us look first at the declaration part :

pcap_parse.py

Listing 4.3 – pcap_parse.py

```
#!/usr/bin/env python3
2 from scapy.all import *

4 import sys
# insert at 1, 0 is the script path (or '' in REPL)
6 sys.path.insert(1, '../../src/')

8 from compr_parser import Parser
from gen_parameters import *
```

```
import pprint
```

— Line 1 allows a simple execution of the program, if the file has the execution rights [6] it is not necessary to type `python3 ./scapy_read.py`, typing `./scapy_read.py` will be enough.

— Line 2 imports all the modules and functions defined in scapy. Scapy is complementary to openSCHC and will help manipulate IP packets [7].

scapy

— Lines 4 to 6 add in Python3 the path to find the openSCHC modules. They are stored in the `src` directory.

compr_parser

— Lines 8 and 9 import two openSCHC modules. `Parser` from the module `compr_parser` transforms a packet into a Field Description. Module `gen_parameters` contains the parameters declaration used by openSCHC.

Parser
gen_parameters

— line 11 imports the `pprint` module to offer a more readable display of Python structures.

pprint

```
# rdpcap comes from scapy and loads in our pcap file
packets = rdpcap('trace_coap.pcap')

parser = Parser()
```

— Line 14, the scapy function `rdpcap` opens the pcap file `trace_coap.pcap` containing the capture we will analyze.

rdpcap

— Line 16 creates an instance of the openSCHC Parser class.

Parser

```
# Let's iterate through every packet
for packet in packets:
    hexdump(packet[IPv6])
    packet[IPv6].show()

    if packet[Ether].src == "fa:16:3e:1e:cc:2c":
        direction = T_DIR_DW
    elif packet[Ether].dst == "fa:16:3e:1e:cc:2c":
        direction = T_DIR_UP
    else: # skipping
        break

    print ("Packet direction ", direction)

    parsed = parser.parse (bytes(packet[IPv6]),
                           direction,
                           layers=["IPv6", "UDP"])
```

6. type `chmod +x scapy_read.py`

7. It is better to manually install scapy to get the latest version. See `https://scapy.readthedocs.io/en/latest/installation.html`.

```
    pprint.pprint (parsed)
```

— Line 19, the loop ensures that the scapy structure `packet` will be contains each of the packet records stored in the `pcap` file.

— Lines 20 and 21 display the packet in two different ways [8]. The scapy function `hexdump` returns a hexadecimal and ASCII dump of the packet content. On the contrary, the `show` method disassembles the packet and displays all fields. The following listing gives an example of such a display.

hexdump

show

scapy

```
0000  60 07 51 9F 00 20 11 30 20 01 41 D0 04 04 02 00  `.Q.. .0 .A.....
0010  00 00 00 00 00 00 3A 86 20 01 41 D0 03 02 22 00  ......:. .A...".
0020  00 00 00 00 00 00 13 B3 81 B9 16 33 00 20 9C 8B  ...........3. ..
0030  42 01 9E F8 3E C5 3C 75 73 65 72 2E 61 63 6B 6C  B...>.<user.ackl
0040  2E 69 6F 84 74 69 6D 65                          .io.time
###[ IPv6 ]###
  version   = 6
  tc        = 0
  fl        = 479647
  plen      = 32
  nh        = UDP
  hlim      = 48
  src       = 2001:41d0:404:200::3a86
  dst       = 2001:41d0:302:2200::13b3
###[ UDP ]###
     sport     = 33209
     dport     = 5683
     len       = 32
     chksum    = 0x9c8b
###[ Raw ]###
        load      = 'B\x01\\x9e\\xf8>\\xc5<user.ackl.io\\x84time'
```

— Lines 23 to 30 are specific to SCHC. The pcap file records all the traffic on a specific network, but SCHC requires to affect a direction (up-link or down-link) to a packet. To do so, we look at the source MAC address. If the frames originate from the MAC address given line 23, then the packet is considered as downlink. If the MAC address is not found, the packet is skipped. The direction is printed when the packet is recognized.

— Lines 32 to 34 the SCHC parser is called. It takes as arguments the packet to be parsed (the bytes transform the sketchy structure to a byte array), the direction we just compute and the layers to be parsed. In this example, only IPv6 and UDP headers are parsed.

— line 35 displays the result of the parsing. It is a tuple that contains :

  — the field description consisting of a Python dictionary where the key is the Field ID and its position and the value and array with the field content as a byte array followed by its length.

  — remaining data

  — a notification, if errors occurs during parsing.

```
Packet direction  UP
({('IPV6.APP_IID', 1): [b'\x00\x00\x00\x00\x00\x00\x13\xb3', 64],
  ('IPV6.APP_PREFIX', 1): [b' \x01A\xd0\x03\x02"\x00', 64],
  ('IPV6.DEV_IID', 1): [b'\x00\x00\x00\x00\x00\x00:\x86', 64],
```

8. The capture includes the Ethernet frame. So `packet[IPv6]` limits to IPv6 layers and above, excluding Ethernet header.

```
  ('IPV6.DEV_PREFIX', 1): [b' \x01A\xd0\x04\x04\x02\x00', 64],
  ('IPV6.FL', 1): [b'\x07Q\x9f', 20],
  ('IPV6.HOP_LMT', 1): [b'0', 8, 'fixed'],
  ('IPV6.LEN', 1): [b'/', 16, 'fixed'],
  ('IPV6.NXT', 1): [b'\x11', 8, 'fixed'],
  ('IPV6.TC', 1): [b'\x00', 8],
  ('IPV6.VER', 1): [b'\x06', 4],
  ('UDP.APP_PORT', 1): [b'\x163', 16],
  ('UDP.CKSUM', 1): [b'\xf5\xef', 16],
  ('UDP.DEV_PORT', 1): [b'\x81\xb9', 16],
  ('UDP.LEN', 1): [b'/', 16]},
 b'B\x03\x9e\xf7>\xc4<user.ackl.io\x85other\x05block\xffHLO 009',
 None)
```

4.3.2 Rule Manager

The next step is to find a rule matching the packet description. The program `pcap_match.py`- introduces the Rule Manager to manipulate devices' rules.

pcap_match.py

Rule Manager

Listing 4.4 – pcap_match.py

```
#!/usr/bin/env python3
from scapy.all import *

import sys
# insert at 1, 0 is the script path (or '' in REPL)
sys.path.insert(1, '../../src/')

from compr_parser import Parser
from gen_parameters import *
from gen_rulemanager import RuleManager

import pprint

# rdpcap comes from scapy and loads in our pcap file
packets = rdpcap('trace_coap.pcap')

parser = Parser()

RM = RuleManager()
RM.Add(file="ipv6.json")
RM.Print()
```

pcap_parse.py

This program derives from `pcap_parse.py` previously explained. The differences are :

gen_rulemanager

RuleManager

- Line 10 to import the class `RuleManager` from the module `gen_rulemanager.py`.
- line 19 to create the instance `RM` of the `RuleManager`.
- line 20 to import the `ipv6.json` rule as defined on listing 4.2 on page 50.
- line 21 to print the rules known by the rule manager. Note that the rule manager add fields such as Field Length, Field Direction, Field Position. They can be specified in the JSON file to override the default values.

```
****************************************
Device: None
/-------------------------\
|Rule 5/3             101  |
|---------------+---+--+--+------------------------------+-------------+----------------\
```

```
|IPV6.VER       |  4| 1|BI|                              06|EQUAL        |NOT-SENT        |
|IPV6.TC        |  8| 1|BI|                              01|EQUAL        |NOT-SENT        |
|IPV6.FL        | 20| 1|BI|                          023456|EQUAL        |NOT-SENT        |
|IPV6.LEN       | 16| 1|BI|--------------------------------|IGNORE       |COMPUTE-LENGTH  |
|IPV6.NXT       |  8| 1|BI|                              11|EQUAL        |NOT-SENT        |
|IPV6.HOP_LMT   |  8| 1|BI|--------------------------------|IGNORE       |VALUE-SENT      |
|IPV6.DEV_PREFIX| 64| 1|BI|                fe80000000000000|MATCH-MAPPING|MAPPING-SENT    |
:       .       :  .:. :. :                200141d003022200:      .      :       .        :
|IPV6.DEV_IID   | 64| 1|BI|                00000000000013b3|EQUAL        |NOT-SENT        |
|IPV6.APP_PREFIX| 64| 1|BI|                200141d004040200|MATCH-MAPPING|MAPPING-SENT    |
:       .       :  .:. :. :                fe80000000000000:      .      :       .        :
|IPV6.APP_IID   | 64| 1|BI|                0000000000000002|EQUAL        |NOT-SENT        |
|UDP.DEV_PORT   | 16| 1|BI|--------------------------------|IGNORE       |VALUE-SENT      |
|UDP.APP_PORT   | 16| 1|BI|                            1630|MSB(12)      |LSB             |
|UDP.LEN        | 16| 1|BI|                              00|IGNORE       |COMPUTE-LENGTH  |
|UDP.CKSUM      | 16| 1|BI|                              00|IGNORE       |COMPUTE-CHECKSUM|
\---------------+---+--+--+--------------------------------+-------------+----------------/
```

4.3.3 Rule Matching

FindRuleFromPacket

The program `pcap_match.py` continues with rule matching, calling the method `FindRuleFromPacket` from the `RuleManager` object, as specified at lines 42 to 48.

RuleManager

Listing 4.5 – pcap_match.py

```
# Let's iterate through every packet
for packet in packets:
    #hexdump(packet[IPv6])
    #packet[IPv6].show()

    if packet[Ether].src == "fa:16:3e:1e:cc:2c":
        direction = T_DIR_DW
    elif packet[Ether].dst == "fa:16:3e:1e:cc:2c":
        direction = T_DIR_UP
    else: # skipping
        break

    print ("Packet direction ", direction)

    parsed = parser.parse (bytes(packet[IPv6]),
                           direction,
                           layers=["IPv6", "UDP"])
    pprint.pprint (parsed)

    if parsed[0] != None:
        rule = RM.FindRuleFromPacket(pkt=parsed[0],
                                     direction=direction,
                                     failed_field=True)

        print ("Rule for packet")
        pprint.pprint(rule)
```

This method takes 3 arguments :

— The parsed packet (i.e. the first element of the tuple returned by the parser.

— The direction.

failed_field

— A debug option `failed_field` which displays the matching process and highlights non-matching fields.

We can see from the trace that no packets in the pcap file match the rule.

```
Packet direction  DW
({('IPV6.APP_IID', 1): [b'\x00\x00\x00\x00\x00\x00\x13\xb3', 64],
  ('IPV6.APP_PREFIX', 1): [b' \x01A\xd0\x03\x02"\x00', 64],
  ('IPV6.DEV_IID', 1): [b'\x00\x00\x00\x00\x00\x00:\x86', 64],
  ('IPV6.DEV_PREFIX', 1): [b' \x01A\xd0\x04\x04\x02\x00', 64],
  ('IPV6.FL', 1): [b'\nE\xf8', 20],
  ('IPV6.HOP_LMT', 1): [b'@', 8, 'fixed'],
  ('IPV6.LEN', 1): [b'\x1f', 16, 'fixed'],
  ('IPV6.NXT', 1): [b'\x11', 8, 'fixed'],
  ('IPV6.TC', 1): [b'\x00', 8],
  ('IPV6.VER', 1): [b'\x06', 4],
  ('UDP.APP_PORT', 1): [b'\x163', 16],
  ('UDP.CKSUM', 1): [b'=\x12', 16],
  ('UDP.DEV_PORT', 1): [b'\x81\xb9', 16],
  ('UDP.LEN', 1): [b'\x1f', 16]},
 b'bE\x9e\xf4>\xc1\xff2023-04-06 10:10',
 None)
{'FID': 'IPV6.VER', 'FL': 4, 'FP': 1, 'DI': 'BI', 'TV': b'\x06', 'MO': 'EQUAL', 'CDA': 'NOT-SENT'}
{'FID': 'IPV6.TC', 'FL': 8, 'FP': 1, 'DI': 'BI', 'TV': b'\x01', 'MO': 'EQUAL', 'CDA': 'NOT-SENT'}
rule 5/3: field IPV6.TC  does not match TV=b'01' FV=b'00' rlen=8 flen=8 arg=None
Rule for packet
None
```

By examining the trace, we can see that there is a matching for `IPV6.VER` field, but the next field `IPV6.TC` do not match, therefore rule `5/3` is not selected.

Question 4.3.1: Rule modification

Adapt the rule stored in `ipv6.json` file (see Listing 4.2 on page 50) to allow for at least one downlink packet matching.

Looking at the execution trace 4.3.3, we can step by step modify the rule :

- the TV is 1 and the Field Value (FV) is 0 for the Traffic Class `IPV6.TC`. The first adaption is to change the value in the rule. `IPV6.TC`
- By launching the program after the rule modification, we also need to modify the Flow Label (`IPV6.FL`) value to 0xa45f8. The value has to be in decimal in the JSON file, it corresponds to 673272. `IPV6.FL`
- Then, the `DEV.PREFIX` prefix must be added or modified to the matching list. We choose to add it; this will change the residue size, since 2 bits are now necessary to code the 3 possible values.
- The `DEV.IID` must also be changed to `::3a86`.
- The `IPV6.APP_IID` needs also to be changed to `::13b3`. `IPV6.APP_IID`
- The same adaption has to be done for the Application address.
- Finally, in the `UPD.APP_PORT` field, the `TV` value should be changed to `5680` so that the 12 MSB match.

This leads to the rule given in Listing 4.6.

Listing 4.6 – ipv6-sol

```
****************************************
2 Device: None
/------------------------\
```

```
|Rule 5/3            101   |
|---------------+---+--+--+------------------------------+-------------+----------------\
|IPV6.VER       |  4| 1|BI|                            06|EQUAL        |NOT-SENT        |
|IPV6.TC        |  8| 1|BI|                            00|EQUAL        |NOT-SENT        |
|IPV6.FL        | 20| 1|BI|                        0a45f8|EQUAL        |NOT-SENT        |
|IPV6.LEN       | 16| 1|BI|------------------------------|IGNORE       |COMPUTE-LENGTH  |
|IPV6.NXT       |  8| 1|BI|                            11|EQUAL        |NOT-SENT        |
|IPV6.HOP_LMT   |  8| 1|BI|------------------------------|IGNORE       |VALUE-SENT      |
|IPV6.DEV_PREFIX| 64| 1|BI|              fe80000000000000|MATCH-MAPPING|MAPPING-SENT    |
:       .       : . :. :. :              200141d003022200:      .      :       .        :
:       .       : . :. :. :              200141d004040200:      .      :       .        :
|IPV6.DEV_IID   | 64| 1|BI|              0000000000003a86|EQUAL        |NOT-SENT        |
|IPV6.APP_PREFIX| 64| 1|BI|              200141d004040200|MATCH-MAPPING|MAPPING-SENT    |
:       .       : . :. :. :              200141d003022200:      .      :       .        :
:       .       : . :. :. :              fe80000000000000:      .      :       .        :
|IPV6.APP_IID   | 64| 1|BI|              00000000000013b3|EQUAL        |NOT-SENT        |
|UDP.DEV_PORT   | 16| 1|BI|------------------------------|IGNORE       |VALUE-SENT      |
|UDP.APP_PORT   | 16| 1|BI|                          1630|MSB(12)      |LSB             |
|UDP.LEN        | 16| 1|BI|                            00|IGNORE       |COMPUTE-LENGTH  |
|UDP.CKSUM      | 16| 1|BI|                            00|IGNORE       |COMPUTE-CHECKSUM|
\---------------+---+--+--+------------------------------+-------------+----------------/
```

We can see that we match half of the packets, which correspond to the downlink messages. There is no rule that matches the uplink messages.

Question 4.3.2: Bi-directional rule

How to make that rule bidirectional ? You can add a Direction Indicator (key "DI" in openSCHC format, and a value "UP" or "DW") to force the matching in a specific direction.

4.4 Hands-on

4.4.1 Let's compress

We have the packet, we have the rule, so we can have the compression.

Listing 4.7 – pcap_compress.py

```
#!/usr/bin/env python3
from scapy.all import *

import sys
# insert at 1, 0 is the script path (or '' in REPL)
sys.path.insert(1, '../../src/')

from compr_parser import Parser
from gen_parameters import *
from gen_rulemanager import RuleManager
from compr_core import Compressor

import pprint

# rdpcap comes from scapy and loads in our pcap file
packets = rdpcap('trace_coap.pcap')

parser = Parser()
```

```
20 RM = RuleManager()
   RM.Add(file="ipv6-sol-bi-fl.json")
22 RM.Print()

24 compress = Compressor()

26 # Let's iterate through every packet
   for packet in packets:
28     #hexdump(packet[IPv6])
       #packet[IPv6].show()
30
       if packet[Ether].src == "fa:16:3e:1e:cc:2c":
32         direction = T_DIR_DW
       elif packet[Ether].dst == "fa:16:3e:1e:cc:2c":
34         direction = T_DIR_UP
       else: # skipping
36         break

38     print ("Packet␣direction␣", direction)

40     parsed = parser.parse (bytes(packet[IPv6]),
                              direction,
42                            layers=["IPv6", "UDP"])
       pprint.pprint (parsed)
44
       if parsed[0] != None:
46         rule = RM.FindRuleFromPacket(pkt=parsed[0],
                                        direction=direction,
48                                      failed_field=True)

50         print ("Rule␣for␣packet")
           pprint.pprint(rule)
52
           if rule:
54             SCHC_pkt = compress.compress(rule=rule,
                                            parsed_packet=parsed[0],
56                                          data= parsed[1],
                                            direction=direction)
58
               print("SCHC␣packet␣in␣hex")
60             SCHC_pkt.display()

62             print("SCHC␣packet␣in␣binary")
               SCHC_pkt.display(format="bin")
```

Listing 4.7 on the previous page extends the previous program, given Listing 4.5 on page 55 and introduces the SCHC compression :

— Line 11 imports the module, compr_core.py

— Line 24 creates an instance of the `Compressor`, Compressor

— in lines 53 to 63, if a rule matches the packet description, the `compress` method compress compress

returns the SCHC packet. It takes the rule, the packet description, the remaining data after the parsing, and the direction.

BitArray The result is `SCHC_pkt` stored in a openSCHC structure called the `BitArray`. This class helps to manipulate bit-per-bit a binary buffer. Lines 60 and 63 display the content, respectively, in hexadecimal and binary. gen_bitarray.py

```
SCHC packet in hex
a81303726c48b3df07d8bfe646064665a60685a606c4062607462600/219
SCHC packet in binary
101010000001001100000011011100100110110001001000101100111101
111100000111110110001011111111100110010001100000011001000110
011001011010011000000110100001011010011000000110110001000000
01100010011000000111010001100010011000000000/219
------------------------------------------------------------
------------------------------------------------------------
------------------------------------------------------------
---------------------------------------
```

The hexadecimal representation is compact and is followed by the length in bits (`/219`) but may be difficult to manipulate and to identify the padding bits at the end.

The binary representation is followed by a sequence of `-`. They give exactly where the buffer stop. It is easier to see in this trace that the last 4 bits equal to `0` are padding bits.

Question 4.4.1: SCHC packet

Identify in the trace 10.5 on page 134, the different elements that make up a SCHC packet (rule ID, residues, data, and padding). You can add the `verbose=True` argument when calling the compresson method.

4.4.2 Let's decompress

With the compressed SCHC packet and the set of rules, we can perform the opposite operation to recover the header.

Listing 4.8 – pcap_decompress.py

```
#!/usr/bin/env python3
from scapy.all import *

import sys
# insert at 1, 0 is the script path (or '' in REPL)
sys.path.insert(1, '../../src/')

from compr_parser import Parser, Unparser
from gen_parameters import *
from gen_rulemanager import RuleManager
from compr_core import Compressor, Decompressor
```

```
import pprint
import binascii

# rdpcap comes from scapy and loads in our pcap file
packets = rdpcap('trace_coap.pcap')

parser = Parser()
Unparser = Unparser()

RM = RuleManager()
RM.Add(file="ipv6-sol.json")
RM.Print()

compress = Compressor()
decompress = Decompressor()

def show_diff(s1, s2):
    from termcolor import colored

    if len(s1) != len(s2):
        print("size is different")
        return

    differ = False
    for o, c in zip(s1, s2):
        #print(o, c)
        if o == c:
            print(colored(chr(o), "green"), end="")
        else:
            print(colored(chr(o), "red"), end="")
            differ = True
    print()
    if differ:
        print (s2.decode())


# Let's iterate through every packet
for packet in packets:
    #hexdump(packet[IPv6])
    #packet[IPv6].show()

    if packet[Ether].src == "fa:16:3e:1e:cc:2c":
        direction = T_DIR_DW
    elif packet[Ether].dst == "fa:16:3e:1e:cc:2c":
        direction = T_DIR_UP
    else: # skipping
        break

    print ("Packet direction ", direction)

    parsed = parser.parse (bytes(packet[IPv6]),
                           direction,
```

```
65                                 layers=["IPv6", "UDP"])
    pprint.pprint (parsed)
67
    if parsed[0] != None:
69      rule = RM.FindRuleFromPacket(pkt=parsed[0],
                                     direction=direction,
71                                   failed_field=True)

73      print ("Rule for packet")
        pprint.pprint(rule)
75
        if rule:
77          SCHC_pkt = compress.compress(rule=rule,
                                         parsed_packet=parsed[0],
79                                       data= parsed[1],
                                         direction=direction,
81                                       verbose=True)

83          print("SCHC packet in hex")
            SCHC_pkt.display()
85
            print("SCHC packet in binary")
87          SCHC_pkt.display(format="bin")

89          field_description = decompress.decompress(rule=rule,
                                                      schc=SCHC_pkt,
91                                                  direction=direction)
            print(field_description)
93          SCHC_pkt.display(format="bin")
            data = SCHC_pkt.get_remaining_content()
95          print ("payload:", binascii.hexlify(data))

97          pkt = Unparser.unparse(header_d=field_description,
                                   data=data,
99                                 direction=direction)

101         show_diff(binascii.hexlify(bytes(packet)[14:]),
                      binascii.hexlify(bytes(pkt)))
```

This programm adds :

compr_core — Line 11, the import of the class `compr_core` from teh `compr_core` module. Decompressor

— Line 27 creates an instance of the class.

show_diff — Lines 29 to 46 the function show_diff will help compare to hexadecimal sequence to enlighten the differences.

— lines 89 to 91, the decompression method is called. It contains the rule, the SCHC packet, and the direction. The result is a field description. We can notice that fields tagged with the `compute-*` CDA do not contain values, since at this point it is impossible to compute the checksum or the length. compute-*

```
{('IPV6.VER', 1): [b'\x06', 4],
 ('IPV6.TC', 1): [b'\x00', 8],
 ('IPV6.FL', 1): [b'\nE\xf8', 20],
 ('IPV6.LEN', 1): ('LLLL', 16),
 ('IPV6.NXT', 1): [b'\x11', 8],
 ('IPV6.HOP_LMT', 1): [b'@', 8],
 ('IPV6.DEV_PREFIX', 1): [b' \x01A\xd0\x04\x04\x02\x00', 64],
 ('IPV6.DEV_IID', 1): [b'\x00\x00\x00\x00\x00\x00:\x86', 64],
 ('IPV6.APP_PREFIX', 1): [b' \x01A\xd0\x03\x02"\x00', 64],
 ('IPV6.APP_IID', 1): [b'\x00\x00\x00\x00\x00\x00\x13\xb3', 64],
 ('UDP.DEV_PORT', 1): [b'\x81\xb9', 16],
 ('UDP.APP_PORT', 1): [b'\x163', 16],
 ('UDP.LEN', 1): ('LLLL', 16),
 ('UDP.CKSUM', 1): ('CCCC', 16)}
```

— On line 93, the bitbuffer `SCHC_pkt` is displayed in binary form. This time, the second line is made up of = and - characters. The - indicates the end of the buffer and the =, the bits read during the decompression.

```
10101000000010011000000011011100100110110001001000101100111101
1111000001111101100010111111111001100100011000000011001000110
011001011010011000000110100001011010011000000110110001000000
01100010011000000111010001100010011000000000/219
=================================---------------------------
------------------------------------------------------------
------------------------------------------------------------
---------------------------------------
```

get_reamaining_content — Line 94, the method `BitBuffer` extracts the bytes from the bitbuffer and, therefore, removes the padding bits at the end. BitBuffer

— Lines 97 to 99 generate a packet from its field description and payload.

— Lines 101 and 102 compare the original packet (without the 14 bytes of the Ethernet header) with the decompress packet. The display is green, indicating that all compression has not altered bytes.

Question 4.4.2: Compression ratio

In this example, what is the compression ratio of the IPv6/UDP header ?

4.5 Destructive compression

In the rule given in Listing 4.2 on page 50, we are sending all unknown fields such as the IPv6 hop limit or the device UDP port number and we fixed a specific value for the IPv6 Flow label. These values are usually not known when the rule is defined.

To increase the compression rate, in some specific cases, the decompression may not recover the initial values.

4.5.1 Hop Limit

The Hop Limit field can be elided, in both directions :

— On the up-link, the device is at one hop of the of the SCHC core which acts as a router and there is no risk of loop, so the core SCHC can generate a value, for instance 30.

— On the downlink, the value changes depending on the number of routers the packet will have to cross. But we do not really care for this value. Hop Limit purpose is to limit the impact of routing loops when the packet is sent to the device. In our architecture, the device is the next hop. If we set the Hop Limit to 1, the device must process the packet and cannot forward it.

The decoupling between MO and CDA becomes obvious. If we use the matching operator 'ignore', any value is possible, and if we use the CDA 'not sent', the decompressor will use the value stored in the Target Value. By setting it to 30 for uplink traffic, the packet can reach its destination on the Internet, and by setting it to 1 for downlink traffic, the packet cannot be routed by the device.

Of course, if the device acts as a router, you can choose another value than 1 or send the Hop Limit field by modifying the rule.

4.5.2 Flow Label

Flow label cannot be guessed when the rule is described, they are chosen dynamically by the source, to identify a flow and help the packet processing inside the network. RFC 6437 indicates that the value 0 indicates that the packet is not labeled.

Question 4.5.1: Hop Limit

Modify the previous rule to handle the Flow Label and Hop Limit fields as described in this section. Is the checksum modified ?

4.6 Rule description Optimization

This section analyses each header field value and provides different strategies for describing them in the Rule.

Some field's values are not defined in the standards and are left open to the implementation. Most of these fields will take different values and need different Rules.

The choice of Strategy is not arbitrary but rather a result of understanding the specific application and the hosts' performance. This knowledge enables informed decisions and optimizes compression.

In Chapter 2, a basic description of the header field's values and the standard definition enable the possibility to describe some rules. A deep analysis improves the compression's performance but implies using more Rules.

4.7 CoAP Rules

The compression of CoAP headers is a little more complex than IPv6 and UDP for several reasons :

— Some fields, such as Uri-path, may have a variable length changing from one packet to the other,

Uri-path

— Some fields may also be repeated several times in the header, Uri-path and Uri-query are some examples.

Uri-query

— The Options in CoAP make the header format more flexible depending on the application.

— The Message format in a CoAP request and a CoAP response is different.

For the latter point, we saw that Direction Indicator can be used to define a specific format for each direction, and therefore it is possible to define a single rule that handles CoAP requests and responses [9].

Direction Indicator

4.7.1 Option description

SCHC considers each option as a field and defines a specific Field ID as shown in Table 4.2 on page 67. This description covers the most common fields defined for CoAP. Note the No-Response option which will play an important role in LPWAN networks.

Field ID

No-Response

Generally an option is considered as a single field and which can be repeated several times. That will be the case for Uri-path or Uri-Query. In the rule, Field Position indicates the instance of that particular field, starting from 1 [10].

Field Position

We can notice from Table 4.2 on page 67, that there is no Field ID for the Payload Marker (`0xFF`) used to indicate the end of the option part and the start of the payload. This element does not carry any information. The list of headers in the rule defines the CoAP header.

Payload Marker

4.7.2 Variable Length

With IPv6 and UDP, the Field Length is well known and never vary from one packet to another. It was then possible to indicate the length in bits in the rule, and the residue size

9. However, this is not the only possibility, if Devices and Application play both the role of client and server, another alternative is to define a rule for requests and others for response.

10. The RFC 8724 allows to give position 0 to fields, which means that the position is not important, and the decompressor selects the order. This can be useful for the Uri-query fields.

Uri-query

was known by each end.

In CoAP, two types of fields have a variable length :

token
Token Length

— token where the length is specified in the header, we need to avoid incoherence between the information stored in the Token Length field and the length of the token value itself.

— Most of the options where the length is coded in the byte preceding the value. As we saw, CoAP developed its own coding of length. Values between 0 and 12 are directly stored in the preceding byte, for higher length values, special values are used and the length is stored before the data.

The options processing lightens the compression process of SCHC. It does not work directly on the packet data but on an abstract representation of a field. The way CoAP codes the length is not seen by the SCHC compression process.

openSCHC

In openSCHC, the parsing and unparsing processes are responsible for the conversion.

The Field Length can contain a function instead of a specific value :

tkl

— tkl to indicate that the Token Length is given by the Token Length field.

var

— var to indicate that the length is not known in the rule definition and should be sent in the residue. This function specifies that the unit is in Bytes and the length on the residue is coded on 4 bits for values less than 15. Value 15 indicates that the next byte is used to code the length.

Let us take an example. Suppose that we have to code the element `ichthyofauna` in an Uri-path option. Since it has 13 letters, the CoAP coding will be `0xΔd 0x00 0x69 0x63 0x68 0x74 0x68 0x79 0x6f 0x66 0x61 0x75 0x6e 0x61`.

The first nibble is the delta from the previous CoAP option. The second nibble `d` indicates a length coded on 1 byte, the second byte is this length minus 13, then the characters of the URI path element.

If this option is declared variable as a rule, the residue will be `0xd6 0x96 0x36 0x87 0x46 0x87 0x96 0xf6 0x66 0x17 0x56 0xe6 0x1_`. The sequence is decayed to 4 bits to introduce the length.

4.8 Hand-on

We are going to include CoAP compression on the capture `trace_coap.pcap`.

Question 4.8.1: Transactions

How many type of CoAP transactions can you identify ?

Question 4.8.2: Token Size and Value

What is the size and value of the token ?

Question 4.8.3: Code

What are the code values, are they related to a direction ?

Question 4.8.4: Options

What are the options for the different types of messages.

We choose to create two rules to compress each type of request and their associated response based on the IPv6 UDP compression we just studied.

Question 4.8.5: CoAP Compression

Define the set of rules (with RuleID 5/3 and 6/3), to compress both IPv6/UDP/CoAP sessions.

Question 4.8.6: Residue identification

In compress rule, identify the different elements that make up the compression residue.

Question 4.8.7: Padding

Do these rules (see correction on page 137) introduce padding bits ?

Question 4.8.8: Padding suppression

Propose some solutions to reduce the impact of the padding.

RFC 8824	openSCHC	RFC 9363 (YANG DM)	Length (bits)	Position
CoAP Version	COAP.VER	fid-coap-version	2	1
CoAP Type	COAP.TYPE	fid-coap-type	2	1
CoAP Token Length	COAP.TKL	fid-coap-tkl	4	1
CoAP Code	COAP.CODE	fid-coap-code	8	1
CoAP Message ID	COAP.MID	fid-coap-mid	16	1
CoAP Token	COAP.TOKEN	fid-coap-token	tlk	1
CoAP if-match	COAP.If-Match	fid-coap-option-if-match	var	*
CoAP Uri-host	COAP.Uri-Host	fid-coap-option-uri-host	var	1
CoAP Etag	COAP.ETag	fid-coap-option-etag	var	*
CoAP If-None-Match	COAP.If-None-Match	fid-coap-option-if-none-match	0	1
CoAP Observe	COAP.Observe	fid-coap-option-observe	var	1
CoAP Uri-Port	COAP.Uri-Port	fid-coap-option-uri-port	var	1
CoAP Location-Path	COAP.Location-Path	fid-coap-option-location-path	var	*
CoAP Uri-Path	COAP.Uri-Path	fid-coap-option-uri-path	var	*
CoAP Content-Format	COAP.Content-Format	fid-coap-option-content-format	var	1
CoAP Max-Age	COAP.Max-Age	fid-coap-option-max-age	var	1
CoAP.Uri-Query	COAP.Uri-Query	fid-coap-option-uri-query	var	*
CoAP Max-Age	COAP.Max-Age	fid-coap-option-max-age	var	1
CoAP Accept	COAP.Accept	fid-coap-option-accept	var	1
CoAP Location-Query	COAP.Location-Query	fid-coap-option-location-query	var	1
CoAP Block2	COAP.Block2	fid-coap-option-block2	var	1
CoAP Block1	COAP.Block1	fid-coap-option-block1	var	1
CoAP Size2	COAP Size2	fid-coap-option-size2	var	1
CoAP Proxy-Uri	COAP.Proxy-Uri	fid-coap-option-proxy-uri	var	1
CoAP Proxy-Scheme	COAP.Proxy-Scheme	fid-coap-option-proxy-scheme	var	1
CoAP Size1	COAP.Size1	fid-coap-option-size1	var	1
CoAP.No-Response	COAP.No-Response	fid-coap-option-no-response	8	1
CoAP OSCORE_flags		fid-coap-option-oscore-flags	var	1
CoAP OSCORE_piv		fid-coap-option-oscore-piv	var	1
CoAP OSCORE_kid		fid-coap-option-oscore-kid	var	1
CoAP OSCORE_kidctx		fid-coap-option-oscore-kidctx	var	1

TABLE 4.2 – Field ID definition in different implementations

5. Networking

[1]

In the previous chapter, we saw how to compress and decompress locally packets to generate SCHC messages. OpenSCHC uses scapy to access the network to receive or send packets.

In this chapter, we will see :

- how to configure a node to assign an IPv6 to SCHC devices and allow the core SCHC to process them.
- Set up the state machine to process packets.
- Retrieve packets to be compressed.
- Send decompressed packets on the network.

We will run SCHC on the CORE emulator[2] to create a basic topology, as shown in Figure 5.1 on the facing page. CORE

The system is composed of three virtual machines :

- *App* located on the regular internet,
- *Core* located at the boundary between the regular internet and the constrained network,
- *Device* located on the constrained network.

In the first scenario, the *App* will ping the *Device* and get the answer.

1. The integration has to be done, but now, to run the CORE emulator, first start `sudo /opt/core/venv/bin/core-daemon` in a shell (root password is `SCHCisGOOD+`), then on another shell, run `core-gui`. From core emu open click on `File>Open>schc-ping.xml`. Start the emulation by clicking the green arrow. Open a shell on each virtual machine by double clicking on the icon. Change the directory to `/home/openschc/openschc/example/MOOC-2` to get the files.

2. `https://coreemu.github.io/core/index.html`

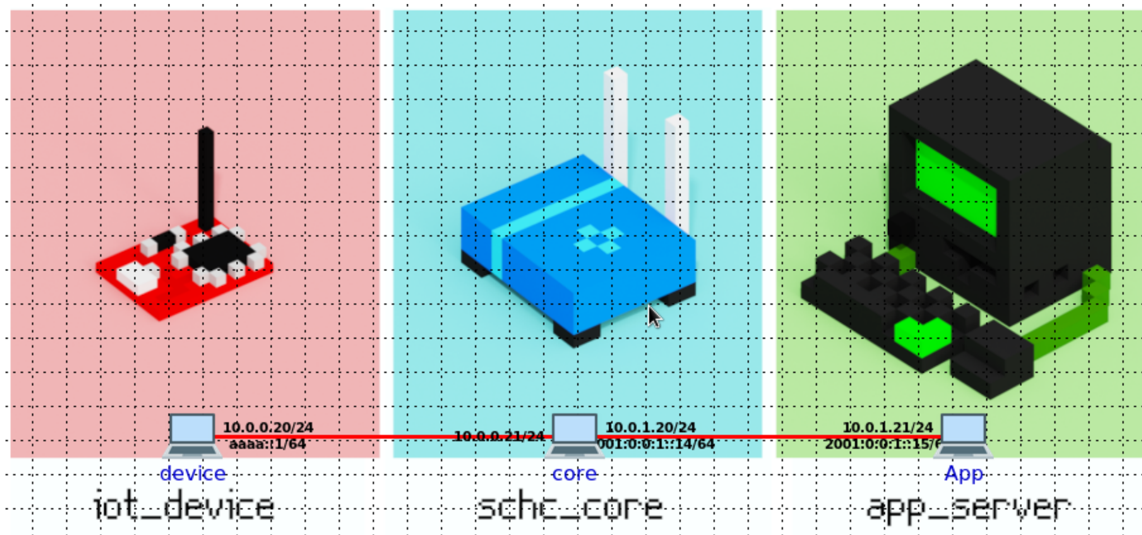


FIGURE 5.1 – Simple topology

5.1 Running the scenario

The network configuration is already configured. The *Core* and the *App* will be connected in IPv6. They can be anywhere on the Internet, but for simplicity the both machines will share the `2001:0:0:1::/64` prefix. The core will have `::14` IID and the *App* will have `::15` IID.

The *Device* and the *Core* are using an IPv4 network and the *Device* is configured with `10.0.0.20/24` address. The two machines exchange SCHC messages, carried in an IPv4 tunnel.

The `aaaa::/64` prefix is allocated to the devices accessible with SCHC. We will allocate `::1` IID to the *Device*. It does not appear in the configuration, since the openSCHC will manage the reachability.

To run the scenario, click on the green arrow of the CORE session ; it will turn red to show that the emulation is running. If you double click on a terminal icon, it will open a terminal where we can execute Linux commands and run openSCHC. Look at the prompt ; it will help you to know on which machine it is running.

We will also open a Code Visual Studio window to test openSCHC, the modification done on the file will be also viewed by the virtual machines.

5.2 First ping

In the *App* terminal, type :

```
# ping6 -i 20  aaaa::1
```

where `-i 20` set the ping interval to 20 seconds and `aaaa::1` is the address of the device. The network is configured to forward the IPv6 packet with `aaaa::/64` destination to the

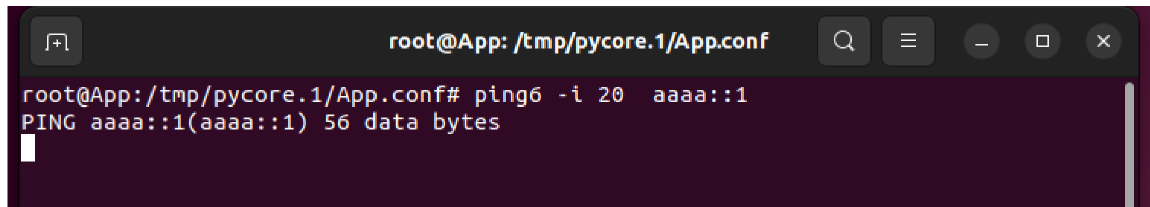


FIGURE 5.2 – First ping

Core. There is no answer since the device is not directly reachable (cf. Figure 5.2).

Now open a shell on the *Core* and start `tcpdump` to look at the incoming traffic : `tcpdump`

```
# tcpdump -lni eth1 ip6
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), snapshot length 262144 bytes
18:04:35.937169 IP6 2001:0:0:1::15 > aaaa::1: ICMP6, echo request, id 22514, seq 13, length 64
18:04:41.058065 IP6 fe80::200:ff:feaa:3 > 2001:0:0:1::14: ICMP6, neighbor solicitation, who has 2001:0:0:1::14, length 32
18:04:41.058107 IP6 2001:0:0:1::14 > fe80::200:ff:feaa:3: ICMP6, neighbor advertisement, tgt is 2001:0:0:1::14, length 24
18:04:56.416282 IP6 2001:0:0:1::15 > aaaa::1: ICMP6, echo request, id 22514, seq 14, length 64
18:05:16.896655 IP6 2001:0:0:1::15 > aaaa::1: ICMP6, echo request, id 22514, seq 15, length 64
18:05:18.944325 IP6 fe80::200:ff:feaa:3 > ff02::2: ICMP6, router solicitation, length 16
```

where :

— `-l` suppresses the buffering and display packets when they arrives,

— `-n` only display numerical values and

— `-i eth1` listen on the `eth1` interface for IPv6 traffic (`ip6` final argument).

We can see that the IPv6 traffic generated by the *App* arrives the *Core*. There is also some other traffic corresponding to the NDP, also based on ICMPv6.

5.3 Compressing traffic on *Core*

5.3.1 Get ping request packets

ping_packet_descr.py

Now, you can run on the *Core* machine, program `ping_packet_descr.py` located in the `MOOC-2` directory of the github repository. This program will take the incoming ICMPv6 message and display a packet description [3] `MOOC-2`

Listing 5.1 – ping_packet_descr.py

```
import sys
# insert at 1, 0 is the script path (or '' in REPL)
sys.path.insert(1, '../../src/')
from compr_parser import Parser
from gen_parameters import *

from scapy.all import *


import pprint

```

3. Note that the parsing is done on a draft version of the specification, and may evolve with time.

```
P = Parser()
13
def processPkt(pkt):
15    if pkt[Ether].type == 0x86dd and pkt[IPv6].nh == 0x3A: #ICMPv6
          pkt_desc = P.parse(pkt=bytes(pkt)[14:], direction=T_DIR_DW)
17        pprint.pprint(pkt_desc)

19 sniff(prn=processPkt, iface="eth1")
```

This program :

- From line 1 to 5, openSCHC module are imported;
- Line 7, scapy module are imported, followed line 10 of the pprint module for a nicer display of Python structures.
- Parser — Line 12, an instance `P` of the `Parser` is created. compr_parser
- from line 14 to 17, the function `processPkt` is declared. This function will be called by scapy when a packet is received. the `pkt` argument contains the received packet in the scapy format.
  - Line 15 is used to test if the incoming packet is a ICMPv6 messages, first the ethertype indicates an IPv6 packet and the IPv6 Next Header field tells that the upper layer is ICMPv6.
  - Line 16, the parser is called with the incoming packet, note that the format is converted from scapy internal representation to a byte array, and the first 14 bytes corresponding to the Ethernet header are skipped. The direction is also specified; since we are dealing with traffic coming from the Internet to the device, it is a downstream traffic.
  - Line 17, the variable `pkt_desc` contains the result of the previous operation. It consists of the tuple composed of Python dictionary containing all the parsed fields, remaining data if there is one, and finally and error code.
- sniff — Line 19, the link with scapy is done through the function `sniff`, taking as arguments : scapy
  - prn — `prn` which indicate which function to call when a packet is received,
  - iface — `iface` giving the interface on which the traffic is captured[4].

The program returns the following Packet Descriptions :

```
({('ICMPV6.CKSUM', 1): [b'7%', 16],
  ('ICMPV6.CODE', 1): [b'\x00', 8],
  ('ICMPV6.PAYLOAD', 1): [b'@\x00\x00\x00\xfe\x80\x00\x00\x00\x00\x00\x00'
                          b'\x02\x00\x00\xff\xfe\xaa\x00\x02',
                          160],
  ('ICMPV6.TYPE', 1): [b'\x88', 8],
  ('IPV6.APP_IID', 1): [b'\x02\x00\x00\xff\xfe\xaa\x00\x02', 64],
  ('IPV6.APP_PREFIX', 1): [b'\xfe\x80\x00\x00\x00\x00\x00\x00', 64],
  ('IPV6.DEV_IID', 1): [b'\x02\x00\x00\xff\xfe\xaa\x00\x03', 64],
  ('IPV6.DEV_PREFIX', 1): [b'\xfe\x80\x00\x00\x00\x00\x00\x00', 64],
  ('IPV6.FL', 1): [b'\x00', 20],
  ('IPV6.HOP_LMT', 1): [b'\xff', 8, 'fixed'],
  ('IPV6.LEN', 1): [b'\x18', 16, 'fixed'],
  ('IPV6.NXT', 1): [b':', 8, 'fixed'],
```

4. If several interfaces are involved, a Python array can be used.

```
  ('IPV6.TC', 1): [b'\x00', 8],
  ('IPV6.VER', 1): [b'\x06', 4]},
 b'',
 None)
({('ICMPV6.CKSUM', 1): [b'|U', 16],
  ('ICMPV6.CODE', 1): [b'\x00', 8],
  ('ICMPV6.IDENT', 1): [b'\xeb\xa6', 16],
  ('ICMPV6.PAYLOAD', 1): [b'\xf8Z\x8ce\x00\x00\x00\x00\xfd\x04\x0c\x00'
                          b'\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17'
                          b'\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#\$\%\&\'()*+,-./'
                          b'01234567',
                          448],
  ('ICMPV6.SEQNO', 1): [b'.', 16],
  ('ICMPV6.TYPE', 1): [b'\x80', 8],
  ('IPV6.APP_IID', 1): [b'\x00\x00\x00\x00\x00\x00\x00\x15', 64],
  ('IPV6.APP_PREFIX', 1): [b' \x01\x00\x00\x00\x00\x00\x01', 64],
  ('IPV6.DEV_IID', 1): [b'\x00\x00\x00\x00\x00\x00\x00\x01', 64],
  ('IPV6.DEV_PREFIX', 1): [b'\xaa\xaa\x00\x00\x00\x00\x00\x00', 64],
  ('IPV6.FL', 1): [b'\x05kH', 20],
  ('IPV6.HOP_LMT', 1): [b'@', 8, 'fixed'],
  ('IPV6.LEN', 1): [b'@', 16, 'fixed'],
  ('IPV6.NXT', 1): [b':', 8, 'fixed'],
  ('IPV6.TC', 1): [b'\x00', 8],
  ('IPV6.VER', 1): [b'\x06', 4]},
 b'',
 None)
```

Question 5.3.1: ICMPv6 Packets

What is the nature of these two ICMPv6 messages ?

We can note that for these two messages, the parser does not return any payload, the payload is included in the Packet Description with the Field ID `ICMPV6.PAYLOAD`. The reason is that ICMPv6 is a stub protocol and there is no upper-layer protocol.

ICMPV6.PAYLOAD

5.3.2 Generating rule for ICMPv6 Ping Request

ping_packet_rule.py

The `ping_packet_rule.py` will help to define a rule for the Ping Request.

Listing 5.2 – ping_packet_rule.py

```
import sys
# insert at 1, 0 is the script path (or '' in REPL)
sys.path.insert(1, '../../src/')
from compr_parser import Parser
from gen_rulemanager import RuleManager
from gen_parameters import *

from scapy.all import *

import pprint

P = Parser()
RM = RuleManager()
RM.Add(file="icmp.json")
RM.Print()

def processPkt(pkt):
    if pkt[Ether].type == 0x86dd and pkt[IPv6].nh == 0x3A: #ICMPv6
        pkt_desc = P.parse(pkt=bytes(pkt)[14:], direction=T_DIR_DW)
        if pkt_desc[2] is None: # No parsing error
            rule = RM.FindRuleFromPacket(pkt_desc[0], T_DIR_DW, True)
```

```
22             print (rule)

24 sniff(prn=processPkt, iface=["eth1", "lo"])
```

The main differences compared to `ping_packet_descr.py` are located :

ping_packet_descr.py

Rule Manager

— Line 5 where the Rule Manager module is loaded, and

— Lines 20 to 22, if the parsing returns no error, then the Rule Manager is called to find an appropriate rule. The last argument (`True`) shows where the rule matching fails.

Question 5.3.2: Ping Request Compression Rule

Define a compression rule allowing any host on the Internet to ping the device.

n that scenario, the IPv6 prefix and IID of the App have to be sent to the device, other fields can be elided. In the ICMPv6 packet, the Sequence Number, the Identifier and the Payload need to be sent.

The following rule `icmp.json` contains an example of the compression rule.

Listing 5.3 – icmp.json

```
{
    "DeviceID" : "udp:10.0.0.20:8888",
    "SoR" : [
          {
            "RuleID": 6,
            "RuleIDLength": 3,
            "Compression": [
                {"FID": "IPV6.VER", "TV": 6, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.TC",  "TV": 0, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.FL",  "TV": 0, "MO": "ignore","CDA": "not-sent"},
                {"FID": "IPV6.LEN",          "MO": "ignore","CDA": "compute-length"},
                {"FID": "IPV6.NXT", "TV": 58, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.HOP_LMT", "TV" : 255,"MO": "ignore","CDA": "not-sent"},
                {"FID": "IPV6.DEV_PREFIX","TV": "AAAA::/64",
                                        "MO": "equal","CDA": "not-sent"},
                {"FID": "IPV6.DEV_IID", "TV": "::1","MO": "equal","CDA": "not-sent"},
                {"FID": "IPV6.APP_PREFIX",         "MO": "ignore","CDA": "value-sent"},
                {"FID": "IPV6.APP_IID",            "MO": "ignore","CDA": "value-sent"},
                {"FID": "ICMPV6.TYPE",  "TV": 128, "MO": "equal","CDA": "not-sent"},
                {"FID": "ICMPV6.CODE",  "TV": 0,   "MO": "equal","CDA": "not-sent"},
                {"FID": "ICMPV6.CKSUM", "TV": 0,   "MO": "ignore","CDA": "compute-checksum"},
                {"FID": "ICMPV6.IDENT", "TV": 0,   "MO": "ignore","CDA": "value-sent"},
                {"FID": "ICMPV6.SEQNO", "TV": 0,   "MO": "ignore", "CDA": "value-sent"},
                {"FID": "ICMPV6.PAYLOAD", "TV": 0,"MO": "ignore","CDA": "value-sent"}
            ]
          }
    ]
  }
```

In the above example, the `IPV6.VERSION` field is not sent, the value 6 is stored is the rule. The `IPV6.NXT` follows the same behavior, since ICMPv6 is expected, the `TV` is 58.

IPV6.VERSION
IPV6.NXT

ignore/not-sent

Note that `IPV6.FL` and `IPV6.HOP_LMT` use the ignore/not-sent combination. During the rule selection, `FL` is not taken into account, but during decompression the value stored in the rule is used. The reason for this is that IPv6 Flow Label (`IPV6.FL`) can be different from 0 in a ping6 request but a constrained IoT device is not expected to take this field into account.

IPV6.FL
IPV6.HOP_LMT

The IPv6 Hop Limit value cannot be anticipated, it depends on how many routers the packet travels through, and it may even vary from one packet to the next. In our example, the rule ignores the incoming value on compression and rebuilds a hop limit value of 64 on

decompression. This behavior is valid if the device is assumed to not forward the packet any further. If the device is indeed an IP router, then the Hop Limit field value must be sent.

The compression of the address fields is a bit trickier : Indeed, the SCHC rule bears no source or destination address. Instead, SCHC uses device and application fields addresses [5], which map to the source and destination according to the direction of the traffic (uplink or downlink). The rule description can therefore be made independent of the direction.

addresses

OpenSCHC splits the addresses into a 64 bit prefix and a 64 bit IID.

IID

In our scenario, the device address is not sent, since the value is known by both the device and the core SCHC entities. In contrast, we allow any host on the Internet to ping the device [6] : the source address of the ping request is sent in full to the device, and the destination address of the ping reply is sent in full to the core (both are subsumed in the App address in the SCHC rule description).

At the ICMPv6 level, the `Type` field is elided in a direction-selective fashion : the Echo Request value (128) is expected downlink. The ICMPv6 `Code` is expected to be 0 in both directions.

`ICMPv6.IDENT` `ICMPV6.SEQNO`

The identifier (`ICMPv6.IDENT`) and sequence number (`ICMPV6.SEQNO`) are not compressed, they are sent as a residue.

`IPV6.LENGTH` `ICMPV6.CHECKSUM`

The other fields `IPV6.LENGTH` and `ICMPV6.CHECKSUM` are not sent, they are recomputed on the decompression side.

The resulting SCHC packet has the following format :

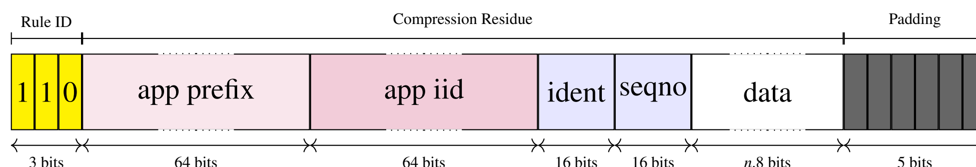


FIGURE 5.3 – ICMPv6 Packet Compressed

The rule ID takes 3 bits, followed by the compression residues, and some padding bits. Note that the SCHC packet contains no data, just a residue that incorporates the ping payload. Since the Rule ID length is 3 bits long and the rest is an integer number of bytes, 5 bits are needed to byte-align the end of the SCHC packet, as needed by the underlying link protocol.

residue

5. the same is true for UDP port numbers.
6. This behavior should be prohibited on a real LPWAN network, to conserve resources. Since we are using an UDP tunnel in our test setup, there is no such issue.

If we run the program, the Neighbor Discovery messages are filtered, since no rule are defined for the link local addresses they uses, but the Ping Request matches rule 6/3.

```
****************************************
Device: udp:10.0.0.20:8888
/-------------------------\
|Rule 6/3              110   |
|---------------+---+--+--+------------------------------+------------+----------------\
|IPV6.VER       |  4| 1|BI|                            06|EQUAL       |NOT-SENT        |
|IPV6.TC        |  8| 1|BI|                            00|EQUAL       |NOT-SENT        |
|IPV6.FL        | 20| 1|BI|                            00|IGNORE      |NOT-SENT        |
|IPV6.LEN       | 16| 1|BI|------------------------------|IGNORE      |COMPUTE-LENGTH  |
|IPV6.NXT       |  8| 1|BI|                            3a|EQUAL       |NOT-SENT        |
|IPV6.HOP_LMT   |  8| 1|BI|                            ff|IGNORE      |NOT-SENT        |
|IPV6.DEV_PREFIX| 64| 1|BI|              aaaa000000000000|EQUAL       |NOT-SENT        |
|IPV6.DEV_IID   | 64| 1|BI|              0000000000000001|EQUAL       |NOT-SENT        |
|IPV6.APP_PREFIX| 64| 1|BI|------------------------------|IGNORE      |VALUE-SENT      |
|IPV6.APP_IID   | 64| 1|BI|------------------------------|IGNORE      |VALUE-SENT      |
|ICMPV6.TYPE    |  8| 1|BI|                            80|EQUAL       |NOT-SENT        |
|ICMPV6.CODE    |  8| 1|BI|                            00|EQUAL       |NOT-SENT        |
|ICMPV6.CKSUM   | 16| 1|BI|                            00|IGNORE      |COMPUTE-CHECKSUM|
|ICMPV6.IDENT   | 16| 1|BI|                            00|IGNORE      |VALUE-SENT      |
|ICMPV6.SEQNO   | 16| 1|BI|                            00|IGNORE      |VALUE-SENT      |
|ICMPV6.PAYLOAD |var| 1|BI|                            00|IGNORE      |VALUE-SENT      |
\---------------+---+--+--+------------------------------+------------+----------------/
{'FID': 'IPV6.VER', 'FL': 4, 'FP': 1, 'DI': 'BI', 'TV': b'\x06', 'MO': 'EQUAL', 'CDA': 'NOT-SENT'}
{'FID': 'IPV6.TC', 'FL': 8, 'FP': 1, 'DI': 'BI', 'TV': b'\x00', 'MO': 'EQUAL', 'CDA': 'NOT-SENT'}
{'FID': 'IPV6.FL', 'FL': 20, 'FP': 1, 'DI': 'BI', 'TV': b'\x00', 'MO': 'IGNORE', 'CDA': 'NOT-SENT'}
{'FID': 'IPV6.LEN', 'FL': 16, 'FP': 1, 'DI': 'BI', 'TV': None, 'MO': 'IGNORE', 'CDA': 'COMPUTE-LENGTH'}
{'FID': 'IPV6.NXT', 'FL': 8, 'FP': 1, 'DI': 'BI', 'TV': b':', 'MO': 'EQUAL', 'CDA': 'NOT-SENT'}
{'FID': 'IPV6.HOP_LMT', 'FL': 8, 'FP': 1, 'DI': 'BI', 'TV': b'\xff', 'MO': 'IGNORE', 'CDA': 'NOT-SENT'}
{'FID': 'IPV6.DEV_PREFIX', 'FL': 64, 'FP': 1, 'DI': 'BI', 'TV': b'\xaa\xaa\x00\x00\x00\x00\x00\x00', 'MO': EQUAL',
        'CDA': 'NOT-SENT'}
rule 6/3: field IPV6.DEV_PREFIX  does not match TV=b'aaaa000000000000' FV=b'fe80000000000000'
          rlen=64 flen=64 arg=None
None
{'FID': 'IPV6.VER', 'FL': 4, 'FP': 1, 'DI': 'BI', 'TV': b'\x06', 'MO': 'EQUAL', 'CDA': 'NOT-SENT'}
{'FID': 'IPV6.TC', 'FL': 8, 'FP': 1, 'DI': 'BI', 'TV': b'\x00', 'MO': 'EQUAL', 'CDA': 'NOT-SENT'}
{'FID': 'IPV6.FL', 'FL': 20, 'FP': 1, 'DI': 'BI', 'TV': b'\x00', 'MO': 'IGNORE', 'CDA': 'NOT-SENT'}
{'FID': 'IPV6.LEN', 'FL': 16, 'FP': 1, 'DI': 'BI', 'TV': None, 'MO': 'IGNORE', 'CDA': 'COMPUTE-LENGTH'}
{'FID': 'IPV6.NXT', 'FL': 8, 'FP': 1, 'DI': 'BI', 'TV': b':', 'MO': 'EQUAL', 'CDA': 'NOT-SENT'}
{'FID': 'IPV6.HOP_LMT', 'FL': 8, 'FP': 1, 'DI': 'BI', 'TV': b'\xff', 'MO': 'IGNORE', 'CDA': 'NOT-SENT'}
{'FID': 'IPV6.DEV_PREFIX', 'FL': 64, 'FP': 1, 'DI': 'BI', 'TV': b'\xaa\xaa\x00\x00\x00\x00\x00\x00', 'MO': EQUAL',
        'CDA': 'NOT-SENT'}
{'FID': 'IPV6.DEV_IID', 'FL': 64, 'FP': 1, 'DI': 'BI', 'TV': b'\x00\x00\x00\x00\x00\x00\x00\x01', 'MO':
        'EQUAL', 'CDA': 'NOT-SENT'}
{'FID': 'IPV6.APP_PREFIX', 'FL': 64, 'FP': 1, 'DI': 'BI', 'TV': None, 'MO': 'IGNORE', 'CDA': 'VALUE-SENT'}
{'FID': 'IPV6.APP_IID', 'FL': 64, 'FP': 1, 'DI': 'BI', 'TV': None, 'MO': 'IGNORE', 'CDA': 'VALUE-SENT'}
{'FID': 'ICMPV6.TYPE', 'FL': 8, 'FP': 1, 'DI': 'BI', 'TV': b'\x80', 'MO': 'EQUAL', 'CDA': 'NOT-SENT'}
{'FID': 'ICMPV6.CODE', 'FL': 8, 'FP': 1, 'DI': 'BI', 'TV': b'\x00', 'MO': 'EQUAL', 'CDA': 'NOT-SENT'}
{'FID': 'ICMPV6.CKSUM', 'FL': 16, 'FP': 1, 'DI': 'BI', 'TV': b'\x00', 'MO': 'IGNORE', 'CDA': 'COMPUTE-CHECKSUM'}
{'FID': 'ICMPV6.IDENT', 'FL': 16, 'FP': 1, 'DI': 'BI', 'TV': b'\x00', 'MO': 'IGNORE', 'CDA': 'VALUE-SENT'}
{'FID': 'ICMPV6.SEQNO', 'FL': 16, 'FP': 1, 'DI': 'BI', 'TV': b'\x00', 'MO': 'IGNORE', 'CDA': 'VALUE-SENT'}
{'FID': 'ICMPV6.PAYLOAD', 'FL': 'var', 'FP': 1, 'DI': 'BI', 'TV': b'\x00', 'MO': 'IGNORE', 'CDA': 'VALUE-SENT'}
{'RuleID': 6, 'RuleIDLength': 3, 'Compression': [{'FID': 'IPV6.VER', 'FL': 4, 'FP': 1, 'DI': 'BI',
...
```

Question 5.3.3: Rule optimization

Modify the previous rule to allow only the App to ping the device, and limit the Sequence Number to 4 bits.

Question 5.3.4: Padding suppression

What will be the Sequence Number residue size to avoid padding at the end of the SCHC message.

5.3.3 The SCHC Machine

In the previous examples, we used several openSCHC functions to parse, find a rule and apply the header compression. It was good to learn how SCHC works, but the code may be

protocol.py simplified by using functions such as `schc_send`, which takes as input a packet, processes the compression, and sends the SCHC packet to the destination. schc_send

In the previous rule, you may have noticed the `DeviceID` key. This is a device identifier that tells the user how to reach the device. In this example, we use an UDP tunnel and the rest of the device identifier contains the IPv4 address of the tunnel end point and the port number. In our case, the IPv4 address is `10.0.0.20` and the port number is 8888. DeviceID

This means that if this rule is selected, the SCHC packet is sent to that destination (cf. figure 5.4).

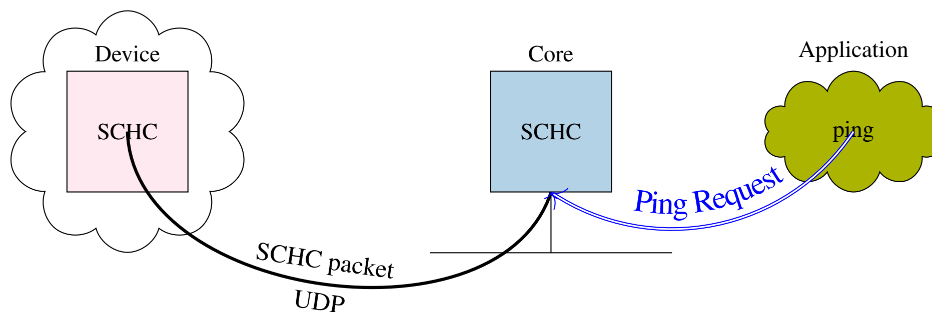


FIGURE 5.4 – SCHC core processing a Ping Request from an Application

The following program allows the compression of Ping Request messages.

Listing 5.4 – ping_core.py

```
import sys
# insert at 1, 0 is the script path (or '' in REPL)
sys.path.insert(1, '../../src/')

from scapy.all import *

import gen_rulemanager as RM
from protocol import SCHCProtocol
from gen_parameters import T_POSITION_CORE

# Create a Rule Manager and upload the rules.

rm = RM.RuleManager()
rm.Add(file="icmp.json")
rm.Print()

def processPkt(pkt):

    scheduler.run(session=schc_machine)
```

```
    if pkt.getlayer(Ether) != None:
        e_type = pkt.getlayer(Ether).type
        if e_type == 0x86dd:
            schc_machine.schc_send(bytes(pkt)[14:], verbose=True)


# Start SCHC Machine
POSITION = T_POSITION_CORE

schc_machine = SCHCProtocol(role=POSITION)
schc_machine.set_rulemanager(rm)
scheduler = schc_machine.system.get_scheduler()


sniff(prn=processPkt, iface="eth1")
```

The principle is the same as the program `ping_packet_rule.py`, but introduce the SCHC machine that will implement the SCHC protocol. The main differences are as follows :

ping_packet_rule.py

- Line 8, the class `SCHCProtocol` is imported. (SCHCProtocol — protocol.py)
- Line 28, the role of the SCHC instance is specified, as a core, in our case. This will specifically determine which traffic is upstream or downstream.
- Line 30, an instance of the SCHC protocol is created
- Line 31, the Rule Manager is associated to that machine
- Line 32, the scheduler used by the SCHC Machine is set to the variable `scheduler`. The scheduler is needed for fragmentation, either to send periodically the fragments or to reassemble them.
- Line 35,as in the previous program, the reception of a packet on the interface `eth1` calls the function `processPkt`. This function is a little bit different :
    - Line 19, a call to the scheduler allows to execute some events, this is transparent to the user.
    - Line 21 to 31, select only IPv6 packets for compression.
    - On line 24, `schc_send` takes the received packet and compresses it. The `verbose` argument allows one to trace the compression process. (schc_send — protocol.py)

Running the program on *Core* gives the following result :

```
# python3 ping_core.py
****************************************
Device: udp:10.0.0.20:8888
/-------------------------\
|Rule 6/3            110  |
|---------------+---+--+--+------------------------------+-------------+----------------\
|IPV6.VER       |  4| 1|BI|                            06|EQUAL        |NOT-SENT        |
|IPV6.TC        |  8| 1|BI|                            00|EQUAL        |NOT-SENT        |
|IPV6.FL        | 20| 1|BI|                            00|IGNORE       |NOT-SENT        |
|IPV6.LEN       | 16| 1|BI|------------------------------|IGNORE       |COMPUTE-LENGTH  |
|IPV6.NXT       |  8| 1|BI|                            3a|EQUAL        |NOT-SENT        |
|IPV6.HOP_LMT   |  8| 1|BI|                            ff|IGNORE       |NOT-SENT        |
|IPV6.DEV_PREFIX| 64| 1|BI|              aaaa000000000000|EQUAL        |NOT-SENT        |
|IPV6.DEV_IID   | 64| 1|BI|              0000000000000001|EQUAL        |NOT-SENT        |
```

```
|IPV6.APP_PREFIX| 64| 1|BI|------------------------------|IGNORE      |VALUE-SENT      |
|IPV6.APP_IID   | 64| 1|BI|------------------------------|IGNORE      |VALUE-SENT      |
|ICMPV6.TYPE    |  8| 1|BI|                            80|EQUAL       |NOT-SENT        |
|ICMPV6.CODE    |  8| 1|BI|                            00|EQUAL       |NOT-SENT        |
|ICMPV6.CKSUM   | 16| 1|BI|                            00|IGNORE      |COMPUTE-CHECKSUM|
|ICMPV6.IDENT   | 16| 1|BI|                            00|IGNORE      |VALUE-SENT      |
|ICMPV6.SEQNO   | 16| 1|BI|                            00|IGNORE      |VALUE-SENT      |
|ICMPV6.PAYLOAD |var| 1|BI|                            00|IGNORE      |VALUE-SENT      |
\---------------+---+--+--+------------------------------+------------+----------------/
rule for compression/no-compression not found, abort sending.
rule for compression/no-compression not found, abort sending.
Use compression rule 6/3
protocol.py, schc_send, core_id:  None device_id:  udp:10.0.0.20:8888 length in bits 623
fragmentation not needed
```

The `verbose` argument shows how the different ICMPv6 messages are processed. In that example, the first two are neighbor discovery messages. Since no rule matches with these messages and that no no-compression rules are defined in the Set or Rule, the messages are discarded. The third message is a Ping Echo Request message and can be compressed with the rule `6/3`. The SCHC packet is 623 bits long and does not require fragmentation [7]. The SCHC packet is sent via UDP to `10.0.0.20`, port 8888.

A `tcpdump` running on *Core* or *Device* visualizes the SCHC packets.

```
# tcpdump -lXni eth0
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
17:22:26.191749 IP 10.0.0.21.23628 > 10.0.0.20.8888: UDP, length 78
        0x0000:  4500 006a a349 4000 4011 8311 0a00 0015  E..j.I@.@.......
        0x0010:  0a00 0014 5c4c 22b8 0056 2c6f c400 2000  ....\L"..V,o....
        0x0020:  0000 0000 2000 0000 0000 0002 a2e6 c026  ...............&
        0x0030:  7e70 5be5 1cca 0000 0000 4fd0 1400 0000  ~p[.......O.....
        0x0040:  0000 2022 2426 282a 2c2e 3032 3436 383a  ..."\$&(*,.02468:
        0x0050:  3c3e 4042 4446 484a 4c4e 5052 5456 585a  <>@BDFHJLNPRTVXZ
        0x0060:  5c5e 6062 6466 686a 6c6e                 \^`bdfhjln
```

5.3.4 On the Device

The device receives the SCHC packet through the UDP tunnel, which may represent an LPWAN network. We need to process the SCHC packet to generate an answer (cf. figure 5.5 on the next page) . There are two approaches :

— Process directly the SCHC packet. This method lacks of universality since it is tailored on the format specified by the rule, but on the other hand, the device does not need to have a implement a full IPv6 stack.

— Decompress the SCHC packet and process the IPv6 packet. This method is generic, but needs a full stack implementation.

We are going to focus on the first approach, where the SCHC packet will be directly manipulated. We also need to expand the rules to take into account the Echo-Reply message.

Echo Reply rule

The only difference between an Echo Request and a Echo Reply message is the Type field, which contains representative value 128 or 129. Rule 6/3 may be duplicated with the new type value and a new Rule ID. This can be a good approach if the device generates ping requests, but in our scenario, the device will only respond to the ping. Therefore, we can

7. The MTU for UDP tunnels is set to 5000 bits.

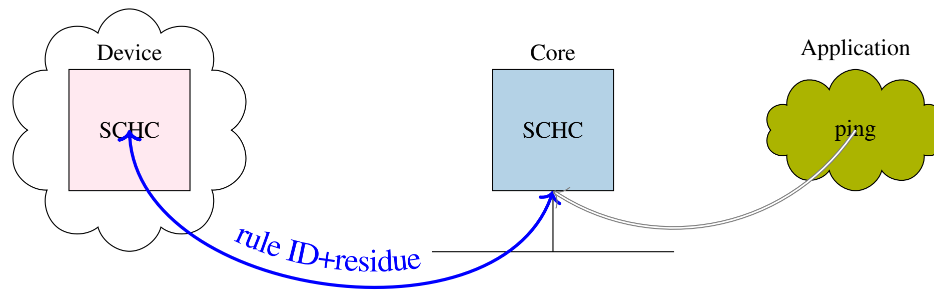


FIGURE 5.5 – SCHC on device processing a Ping Request from an Application and replying

Direction Indicator

use Direction Indicator to fix the value of the Type field in terms of whether it is issued by the Core or sent to the Core.

ICMPV6.TYPE

The rule 6/3 is modified to take into account the direction for ICMPV6.TYPE.

Listing 5.5 – icmp-bi.json

```
{
    "DeviceID" : "udp:10.0.0.20:8888",
    "SoR" : [
        {
            "RuleID": 6,
            "RuleIDLength": 3,
            "Compression": [
                {"FID": "IPV6.VER", "TV": 6, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.TC",  "TV": 0, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.FL",  "TV": 0, "MO": "ignore","CDA": "not-sent"},
                {"FID": "IPV6.LEN",          "MO": "ignore","CDA": "compute-length"},
                {"FID": "IPV6.NXT", "TV": 58, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.HOP_LMT", "TV" : 255,"MO": "ignore","CDA": "not-sent"},
                {"FID": "IPV6.DEV_PREFIX","TV": "AAAA::/64",
                                                 "MO": "equal","CDA": "not-sent"},
                {"FID": "IPV6.DEV_IID", "TV": "::1","MO": "equal","CDA": "not-sent"},
                {"FID": "IPV6.APP_PREFIX",          "MO": "ignore","CDA": "value-sent"},
                {"FID": "IPV6.APP_IID",             "MO": "ignore","CDA": "value-sent"},
                {"FID": "ICMPV6.TYPE",  "TV": 128, "DI" : "DW", "MO": "equal","CDA": "not-sent"},
                {"FID": "ICMPV6.TYPE",  "TV": 129, "DI" : "UP", "MO": "equal","CDA": "not-sent"},
                {"FID": "ICMPV6.CODE",  "TV": 0,   "MO": "equal","CDA": "not-sent"},
                {"FID": "ICMPV6.CKSUM", "TV": 0,   "MO": "ignore","CDA": "compute-checksum"},
                {"FID": "ICMPV6.IDENT", "TV": 0,   "MO": "ignore","CDA": "value-sent"},
                {"FID": "ICMPV6.SEQNO", "TV": 0,   "MO": "ignore", "CDA": "value-sent"},
                {"FID": "ICMPV6.PAYLOAD", "TV": 0,"MO": "ignore","CDA": "value-sent"}
            ]
        }
    ]
}
```

SCHC packet manipulation

The following program echos SCHC messages indetified with Rule ID 6/3 corresponding to an Echo Request message.

Listing 5.6 – ping_dev.py

```
import sys
# insert at 1, 0 is the script path (or '' in REPL)
sys.path.insert(1, '../../src/')
import gen_rulemanager as RM
from gen_parameters import *

```

```
rm = RM.RuleManager()
rm.Add(file="icmp-bi.json")
rm.Print()

import socket
import binascii
import netifaces as ni

addr = ni.ifaddresses('eth0')[ni.AF_INET][0]['addr']

PORT = 8888
deviceID = "udp:"+addr+":"+str(PORT)

print("device ID is", deviceID)

tunnel = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
tunnel.bind (("0.0.0.0", PORT)) # same port as in the DeviceID

while True:
    SCHC_pkt, sender = tunnel.recvfrom(1000)
    print ("SCHC Packet:", binascii.hexlify(SCHC_pkt),
                          "from", sender)
    rule = rm.FindRuleFromSCHCpacket(schc=SCHC_pkt, device=deviceID)
    if rule: # Echo Request Rule
        print ("Rule {}/{}".format(rule[T_RULEID],
                                  rule[T_RULEIDLENGTH]))
        if rule[T_RULEID] == 6 and rule[T_RULEIDLENGTH] == 3:
            tunnel.sendto(SCHC_pkt, sender)
        else:
            print ("Not the Echo Request rule")
    else:
        print ("rule not found")
```

The beginning of the program does not change, SCHC modules are loaded, and then the Rule Manager loads the rule, then :

- Line 14, the module netifaces is loaded to get the IP address associated the `eth0` interface,
- Line 19, the deviceID is built, it is composed of the keywork `udp`, followed by the IP address and the post number. This set is needed since the json file containing the rule indicates the deviceID [8].
- Lines 23 and 24, a socket is created to recieve SCHC messages coming from the Core/
- Lines 26 to 36 and infinite loop process the incoming SCHC packets :
  - Line 25, the program waits for a SCHC packet.

8. the deviceID is needed for the Core since it manages several devices and must know how to reach the device. Since a device in an LPWAN environment talks only with the Core, deviceID can be avoided.

— Lines 26 and 27, a message is displayed to show the content of the SCHC packet and the sender.

FindRuleFromSCHCpacket

— Line 30, a call to `FindRuleFromSCHCpacket` gives the rule, and therefore its ID. Note that the `deviceID` is needed for that lookup.

gen_rulemanager.py

— Lines 31 to 35, if a rule is found and corresponds to the Rule ID `6/3`, the SCHC packet is echoed back to the sender.

With this simple manipulation, the SCHC core will receive a SCHC message through the tunnel. Let us continue and see how the core should process the SCHC packet.

5.3.5 Back to the Core

We now have to take into account the evolution of our architecture. We define a simple core application that takes an ICMPv6 Echo Request, compresses it, and sends it to the device. We have after modification of the rule `6/3` to make the value `ICMPV6.TYPE` dependent on the direction, and the device is sending back a SCHC message. These messages arrive in an IPv4 tunnel on `eth0`.

ICMPV6.TYPE

The following program takes the SCHC packet from the socket and decompresses it and forwards the IPv6 to the application.

Listing 5.7 – ping_core1.py

```
import sys
# insert at 1, 0 is the script path (or '' in REPL)
sys.path.insert(1, '../../src/')

from scapy.all import *

import gen_rulemanager as RM
from protocol import SCHCProtocol
from gen_parameters import T_POSITION_CORE

# Create a Rule Manager and upload the rules.

rm = RM.RuleManager()
rm.Add(file="icmp-bi.json")
rm.Print()

def processPkt(pkt):

    scheduler.run(session=schc_machine)

    if pkt.getlayer(Ether) != None:
        e_type = pkt.getlayer(Ether).type
        if e_type == 0x86dd:
            schc_machine.schc_send(bytes(pkt)[14:])
        elif e_type == 0x0800:
            if pkt[IP].proto == 17 and pkt[UDP].dport == 0x5C4C:
                # got a packet in the socket
                SCHC_pkt, device = tunnel.recvfrom(1000)
```

```
30                  other_end = 'udp:'+device[0]+':'+str(device[1])

32                  origin, full_packet = schc_machine.schc_recv(
                                        schc_packet=SCHC_pkt,
34                                      device_id=other_end,
                                        iface='eth1',
36                                      verbose=True)

38 # Start SCHC Machine
   POSITION = T_POSITION_CORE
40
   schc_machine = SCHCProtocol(role=POSITION)
42 schc_machine.set_rulemanager(rm)
   scheduler = schc_machine.system.get_scheduler()
44 tunnel = schc_machine.get_tunnel()

46 sniff(prn=processPkt, iface=["eth0", "eth1", "lo"])
```

ping_core.py

Compared to `ping_core.py`, there are some small changes :

scapy

sniff

— Line 14, file `icmp-bi.json` is loaded in the rule manager,

— Line 46, `sniff` waits for incoming packet from different interfaces `eth0` is needed to get the packet from the tunnel.

And some code is added in the function `processPkt`. Lines 25 to 36 process IPv4 packets coming from the device tunnel :

— Lines 25 and 26, by checking if an IPv4 packet arrives containing an UDP message to port 0x5C4C,

— Line 28, the `recvfrom` will not block and returns the SCHC message in the `SCHC_pkt` variable.

— Line 30, the address of the sender allows one to create the deviceID which will be used to identify the Set of Rules.

protocol.py

schc_recv

— Lines 32 to 36, the `schc_recv` function is called with several arguments :

  — `schc_packet` contains the SCHC packet recieved from the tunnel,

  — `device_id` contains the device we just built,

  — `iface` tells on which interface the packet should be sent,

  — `verbose` if set to `True` allows to display the packet dump sent on the interface.

  The function returns the device ID and decompresses the packet in the scapy format.

In the *App*, the ping starts to give the following results : [9]

9. You may experience longer delays in the ping, this is due to the very limited numbers of events received by the SCHC machine on this emulation environment. To increase the number of events, open a new shell on

```
# ping6 -i 20 aaaa::1
PING aaaa::1(aaaa::1) 56 data bytes
64 bytes from aaaa::1: icmp_seq=97 ttl=255 time=240 ms
64 bytes from aaaa::1: icmp_seq=98 ttl=255 time=348 ms
64 bytes from aaaa::1: icmp_seq=99 ttl=255 time=137 ms
64 bytes from aaaa::1: icmp_seq=100 ttl=255 time=215 ms
64 bytes from aaaa::1: icmp_seq=101 ttl=255 time=340 ms
64 bytes from aaaa::1: icmp_seq=102 ttl=255 time=104 ms
64 bytes from aaaa::1: icmp_seq=103 ttl=255 time=210 ms
64 bytes from aaaa::1: icmp_seq=104 ttl=255 time=375 ms
```

the core and type `ping 127.0.0.1`.

6. Simple CoAP

In the previous chapter, we learned the compression rule by doing a ping between an Application and a Device. This is not usually the kind of traffic that we expect to see in an LPWAN network. This will drain batteries and will carry no useful information except that the device is on.

In this chapter, we are going to define rules that allow one to communicate in CoAP with a device to exchange information. We will study two scenario :

— The LPWAN scenario. With networks such as LoRaWAN, downlink messages must be kept as rare as possible due to the duty cycle restrictions imposed on the Radio Gateway to operate in unlicensed bands. Therefore, traffic must be triggered by the Device, and acknowledgments cover negative events. In that case, the device acts as a CoAP client and periodically sends a POST to a server running as an application.

duty cycle

POST

— The underwater scenario. In that scenario, a boat queries the resources of the underwater sensors when arriving at a specific area. In that case, the Device acts as a CoAP server ; since we have symmetric communication, with no duty cycle constraints, acknowledgments are possible.

We will keep the same environment as in the previous chapter and we will use files from the `MOOC-3` directory.

6.1 Network architecture

We keep the same architecture as in in previous chapter :

— A *App* will run a CoAP server based on the Python module `aiocoap` [1]. This is the

`aiocoap`

1. `https://github.com/chrysn/aiocoap`

regular distribution, and nothing has been modified to deal with constrained networks.

- A *Device* will act as a client and will send values to the server using the CoAP protocol. As we did for the first version of the ping compression, the device will not implement CoAP completely but will deal with SCHC packets directly.
- A *Core* SCHC between *App* and the *Device* will compress and decompress CoAP messages.

6.2 LPWAN scenario

In this scenario, the device will periodically emit a value captured by different sensors (e.g. temperature, humidity, and pressure). Each measurement will be a resource identified by an URI (respectively `/temp`, `/humi`, `/pres`), coded in CBOR. The CoAP client on the Device, will periodically use a POST to a CoAP server on the App hosts.

NON
No-Response

To avoid positive acknowledgments, the client will use `NON` confirmable messages and the CoAP option `No-Response` will be set to allow only negative notifications [2].

This leads to traffic, where the messages from the devices will have the format shown in Table 6.1 on page 87 with the associated MO and CDA values. The residue is 5 bits long.

The rules for compressing the IPv6 header can be based on the rule used for ICMPv6 compression (see the listing 5.5 on page 79), except that the IPv6 address of the App is known and does not need to be sent as a residue. The IPv6 header is completely elided.

For UDP, the compression will also be complete, since if we suppose that the port numbers are static, every field can be elided.

Figure 6.2 on page 92 shows the SCHC message, if the ruleID is coded on 3 bits, there is no padding, and all the headers are compressed in one byte.

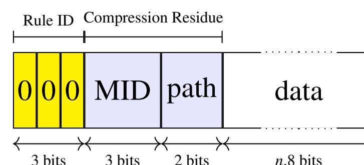


FIGURE 6.1 – IPv6/UDP/CoAP compression

Question 6.2.1: new URI

If the Device includes a new sensor, for example, CO2 level, what is the impact on the description given in the table 6.1 on page 87 ?

2. This option includes a bitmap indicating which class of notification are allowed : `0b_00010010` allows suppresses 2 and 5 and keeps 4 notifications classes.

6.2.1 The Rules

The listing 6.1 presents the two rules 0/3 and 255/8 used in this scenario. The former will compress the uplink messages from the device containing the POST requests, and the latter will compress the possible answer from the CoAP server.

Listing 6.1 – lpwan.json

```
{
    "DeviceID" : "udp:10.0.0.20:8888",
    "SoR" : [{
        "RuleIDValue" : 0,
        "RuleIDLength": 3,
        "Compression": [
            {"FID": "IPV6.VER", "TV": 6, "MO": "equal", "CDA": "not-sent"},
            {"FID": "IPV6.TC",  "TV": 0, "MO": "equal", "CDA": "not-sent"},
            {"FID": "IPV6.FL",  "TV": 0, "MO": "ignore","CDA": "not-sent"},
            {"FID": "IPV6.LEN",          "MO": "ignore","CDA": "compute-length"},
            {"FID": "IPV6.NXT", "TV": 17, "MO": "equal", "CDA": "not-sent"},
            {"FID": "IPV6.HOP_LMT", "TV" : 255,"MO": "ignore","CDA": "not-sent"},
            {"FID": "IPV6.DEV_PREFIX","TV": "AAAA::/64",
                                      "MO": "equal","CDA": "not-sent"},
            {"FID": "IPV6.DEV_IID", "TV": "::1","MO": "equal","CDA": "not-sent"},
            {"FID": "IPV6.APP_PREFIX", "TV": "2001:0:0:1::/64",
                                      "MO": "equal","CDA": "not-sent"},
            {"FID": "IPV6.APP_IID", "TV": "::15",
                                  "MO": "equal","CDA": "not-sent"},

            {"FID": "UDP.DEV_PORT",  "TV": 5388,"MO": "equal",  "CDA": "not-sent"},
            {"FID": "UDP.APP_PORT",  "TV": 5683,"MO": "equal",  "CDA": "not-sent"},
            {"FID": "UDP.LEN",       "TV": 0,   "MO": "ignore", "CDA": "compute-length"},
            {"FID": "UDP.CKSUM",     "TV": 0,   "MO": "ignore", "CDA": "compute-checksum"},

            {"FID": "COAP.VER",  "DI": "BI", "TV": 1,   "MO": "equal","CDA": "not-sent"},
            {"FID": "COAP.TYPE", "DI": "BI", "TV": 1,   "MO": "equal","CDA": "not-sent"},
            {"FID": "COAP.TKL",  "DI": "BI", "TV": 0,   "MO": "equal","CDA": "not-sent"},
            {"FID": "COAP.CODE", "DI": "BI", "TV": 2,   "MO": "equal","CDA": "not-sent"},
            {"FID": "COAP.MID",  "DI": "BI", "TV": 0,               "MO": "MSB", "MO.VAL": 13, "CDA": "LSB"},
            {"FID": "COAP.Uri-Path", "FP": 1, "DI": "UP",
                                "TV": ["temp", "humi", "pres"],
                                "MO": "match-mapping","CDA": "mapping-sent"},
            {"FID": "COAP.Content-Format", "DI": "UP", "TV": 30, "MO": "equal", "CDA": "not-sent"},
            {"FID": "COAP.No-Response",     "DI": "UP", "TV": 2, "MO": "equal", "CDA": "not-sent"}
        ]},
        {
            "RuleIDValue" : 255,
            "RuleIDLength": 8,
            "Compression": [
                {"FID": "IPV6.VER", "TV": 6, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.TC",  "TV": 0, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.FL",  "TV": 0, "MO": "ignore","CDA": "not-sent"},
                {"FID": "IPV6.LEN",          "MO": "ignore","CDA": "compute-length"},
                {"FID": "IPV6.NXT", "TV": 17, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.HOP_LMT", "TV" : 255,"MO": "ignore","CDA": "not-sent"},
                {"FID": "IPV6.DEV_PREFIX","TV": "AAAA::/64",
                                          "MO": "equal","CDA": "not-sent"},
                {"FID": "IPV6.DEV_IID", "TV": "::1","MO": "equal","CDA": "not-sent"},
                {"FID": "IPV6.APP_PREFIX", "TV": "2001:0:0:1::/64",
                                          "MO": "equal","CDA": "not-sent"},
                {"FID": "IPV6.APP_IID", "TV": "::15",
                                      "MO": "equal","CDA": "not-sent"},

                {"FID": "UDP.DEV_PORT",  "TV": 5388,"MO": "equal",  "CDA": "not-sent"},
                {"FID": "UDP.APP_PORT",  "TV": 5683,"MO": "equal",  "CDA": "not-sent"},
                {"FID": "UDP.LEN",       "TV": 0,   "MO": "ignore", "CDA": "compute-length"},
                {"FID": "UDP.CKSUM",     "TV": 0,   "MO": "ignore", "CDA": "compute-checksum"},
                {"FID": "COAP.VER",  "DI": "DW", "TV": 1,   "MO": "equal","CDA": "not-sent"},
                {"FID": "COAP.TYPE", "DI": "DW", "TV": 1,   "MO": "equal","CDA": "not-sent"},
                {"FID": "COAP.TKL",  "DI": "DW", "TV": 0,   "MO": "equal","CDA": "not-sent"},
                {"FID": "COAP.CODE", "DI": "DW", "TV": 132,   "MO": "equal","CDA": "not-sent"},
                {"FID": "COAP.MID",  "DI": "DW", "TV": 0,   "MO": "ignore","CDA": "not-sent"}
            ]}
    ]
}
```

The rule 0/3 totally compresses the IPv6 and UDP headers. This implies a statically allocated port number on the Device CoAP client. CoAP compression is consistent with the description given in the table 6.1 on the next page and the SCHC packet displayed 6.2 on page 92.

Field	Values	compression	MO	CDA
Version	1	*Never change, can be elided.*	equal	not-sent
Type	NON (1)	*To avoid acknowledgments at the transport level, non-confirmable messages will be used.*	equal	not-sent
Token Length	0	*Token are used, since there is only POST, it is not necessary to establish a relation between the request and the response.*	equal	not-sent
Code	POST(2)	*The device will just produce POST requests.*	equal	not-sent
Message ID	0	*Messages ID are used to identify a CoAP message to avoid duplication. Usually the server memorizes the value for 5 minutes. Therefore, a message ID must not be repeated during this period. If 3 bits are used, 7 message ID values are possible, the maximum sending rate is* $300/7 = 43$ *seconds. If the device sends less than one message per minute, 3 bits are enough.* *The value 0 indicates that the 13 most significant bits are set to 0, and only the 3 less significant bits are sent as residue.*	MSB(13)	LSB
uri-path	`/temp`, `/humi`, `/pres`	*The device will issue one of these 3 URIs, a matching list allows sending an index of 2 bits.*	match-mapping	mapping-sent
content-format	CBOR(60)	*The resource is always in CBOR format.*	equal	not-sent
no-response	0000 0010 (2)	*Only 4.0x and 5.0x notification should be sent back by the server. Positive notifications such as 2 2.04 to acknowledge the POST are not sent by the server*	equal	not sent

TABLE 6.1 – CoAP uplink messages

The Rule 255/8 uses destructive compression. None of the CDAs produce residues. If selected, this rule generates an SCHC packet containing only the RuleID. On a constrained device, the nature of the error is not necessary to take a decision. The reception of Rule 255/8 just indicates that an error occurs on the server. In our example, we will change the sending period to simulate energy preservation.

6.2.2 On the device

The device may remain simple; we saw that the overhead of IPv6/UDP/CoAP headers can be reduced to 1 byte for this particular example. It is not necessary, even if it remains possible, to go back to CoAP manipulations. The SCHC message can also be produced by a simple code.

The device sends regularly three measurements related to temperature, humidity, and pressure. The values are coded in CBOR and identified by a specify URL. The class `sensor` contains specific information such as the associated URI, the sending period, and the current value. This class is defined in lines 59 to 81 of `dev-lpwan.py`.

dev-lpwan.py

To ensure different periods for each element, we use an event queue, coded in Python as a list of tuples composed of the sensor class element and its execution time. From this sorted list, the first element is taken, and the system waits until the deadline (lines 90 to 104).

```
    coap_send_measurement(sensor.get_value(), sensor.get_uri())
```

On line 107, the `coap_send_measurement` function takes as argument the value and URI associated with the sensor and sends a compressed SCHC message to the core.

```
MID = 1
16
tunnel = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
18 tunnel.bind(("0.0.0.0", 8888))

20 def coap_send_measurement(value, uri):
       global MID
22
       if uri not in KNOWN_URI:
24         print("unknown URI", uri)
           return None
26
       if type(value) is not bytes: # not cbor
28         value = cbor.dumps(value)

30     uri_idx = KNOWN_URI.index(uri)
       print ("MID", MID, "URI", uri,
32            "(index:", uri_idx, ")",
              "value", binascii.hexlify(value) )
34
       schc_residue = (0x00 & 0b0000_0111) << 5 | \
36                    (MID & 0b0000_0111) << 2 | \
```

```
                    (uri_idx) & 0b0000_0011
38
    if MID == 0b0000_0111:
40      MID = 1
    else:
42      MID += 1

44  schc_pkt = struct.pack("!B", schc_residue) + value
    print ("sending:", binascii.hexlify(schc_pkt))
46  tunnel.sendto(schc_pkt, CORE_SCHC)
```

The function `coap_send_measurement` uses a global variable `MID` (declared line 15) to contain the current value of the Message ID field. Since in SCHC, we compress this field on 3 bits, the value varies between 1 and 7. Lines 39 to 42 are in charge of this incrementation [3].

`KNOWN_URI` is a Python list containing all the possible values accepted by the SCHC rule, in the same position as in the rule. This allows to compute the index `uri_idx` (line 30).

The `schc_residue` (lines 35 to 37) is computed by shifting the RuleID (`0/3`), the 3 less significant bits of the Message ID and the 2 bits of the URI index.

The CBOR is concatenated to the CBOR value (line 44) and sent to the core SCHC through an UDP tunnel (line 46).

The following listing shows the SCHC messages sent by the device.

```
[root@device MOOC-3]$ python3 dev-lpwan.py
sleeping 1 sec.
MID 1 URI temp (index: 0 ) value b'14'
sending: b'0414'
sleeping 28 sec.
MID 2 URI pres (index: 2 ) value b'1903ef'
sending: b'0a1903ef'
sleeping 9 sec.
MID 3 URI humi (index: 1 ) value b'1835'
sending: b'0d1835'
sleeping 21 sec.
MID 4 URI temp (index: 0 ) value b'15'
sending: b'1015'
sleeping 58 sec.
```

We will look at the function `wait_ack` later, let us now focus on how the packet is proceeded in the core SCHC.

6.2.3 The decompression

The `core.py` program is very close to the one used for ping compression. The only difference comes from the use of rule `lpwan.json` by the Rule Manager.

core.py

3. Value 0 for Message ID is forbidden, so it is not possible just to take the less significant bit of a variable.

After decompression, the core sends the IPv6/UDP/CoAP messages to the App.

```
[root@core examples]$ tcpdump -nli eth1 udp port 5683
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), snapshot length 262144 bytes
16:29:20.871998 IP6 aaaa::1.5388 > 2001:0:0:1::15.5683: UDP, length 18
16:29:28.876123 IP6 aaaa::1.5388 > 2001:0:0:1::15.5683: UDP, length 17
```

6.2.4 aiocoap

In this part, we are going to run a regular CoAP server written in Python [4]. The code is named `coap-server.py` and is inspired by the server example from the distribution.

coap-server.py

Listing 6.2 – coap-server.py

```python
import logging
import binascii
import asyncio

import aiocoap.resource as resource
import aiocoap

import cbor2 as cbor
```

aiocoap

Resource

The program imports modules `aiocoap` and `cbor`. This defines a class `Resource`, from which the `sensor_reading` class inherit. This class is used to manipulate resources sent by the Device.

Listing 6.3 – coap-server.py

```python
class sensor_reading(resource.Resource):
    async def render_post(self, request):

        print ("Request␣URI:", request.get_request_uri())
        print ("Content-format:", request.opt.content_format)
        print ("Payload:", binascii.hexlify(request.payload))


        return aiocoap.Message(code=aiocoap.CHANGED)
```

aiocoap

render_

`aiocoap` defines a Python method for every CoAP method, by concatenating the CoAP method name to the `render_` keyword. Here, line 36, a Python method is declared for the CoAP method POST [5].

Here, we process only POST ; other methods will generate an error notification. Argument `request` contains rich information regarding the received CoAP request. The method displays the full URI from different information in the request. For example :

```
Request URI: coap://[2001:0:0:1::15]/temp
```

4. `aiocoap` can be installed through the pip3 command.

aiocoap

5. Possible methods are GET, POST ,PUT, DELETE, FETCH, PATCH, iPATCH, they are defined in the `aiocoap/numbers/codes.py`.

It is also possible to access the CoAP option, through the `opt` structure. In the example, line 39, the *Content-format* option is displayed. It is useful to know this to process the payload correctly.

In the end, the Python method returns a CHANGED (`2.04`) indicating that the resource processing has been performed correctly.

Once the class to process the resource has been defined, it must be linked to the URI, and the server has to be started. This is done in the `main` function.

Listing 6.4 – coap-server.py

```
50 def main():
       # Resource tree creation
52     root = resource.Site()

54     # add resource processing, /proxy is not used here, see comments in generic_sen
       root.add_resource(['temp'], sensor_reading())
56     #root.add_resource(['pres'], sensor_reading())
       root.add_resource(['humi'], sensor_reading())
58
       # associate resource tree and socket
60     asyncio.Task(aiocoap.Context.create_server_context(root))

62     # let's go forever
       asyncio.get_event_loop().run_forever()
64
   if __name__ == "__main__":
66     main()
```

The process is simple. line 52 creates a site object associated with root (/). Then, the different URI are added to that site (lines 55 to 57). `add_resource` takes two arguments. The first is an array that contains all elements of the path [6]. The second argument is an instance of the class (`sensor_reading`) we previously declared.

add_resource aiocoap

render_post

In our case, we use the same class to manage all URIs, which means that `render_post` will have to distinguish between them using the *uri-path* options.

On line 60, the CoAP server is launched, associated with the `root` site we just created. By default, the server listens on all interfaces and on port 5683 [7].

Finally, the call, line 63 launch the server.

Since `/pres` has been commented on, the server only knows the URIs `/temp` and `/humi`. Since the CoAP option *no_response* is set to the value 0x02, all notifications issued by `render_post` are filtered. Only the notification `4.04` will be sent back to the device.

6. Here, only one level is needed.

7. Adding a `bind` argument allows to change de defaults parameters. `asyncio.Task(aiocoap.Context.create_server_context(root), bind=("10.35.131.225", 15386))`

The following capture shows the traffic between the Device and the Application. Not all the CoAP requests incoming from the Device produce a response message.

```
# tcpdump -nli eth1 udp port 5683
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), snapshot length 262144 bytes
17:12:52.968024 IP6 aaaa::1.5388 > 2001:0:0:1::15.5683: UDP, length 18
17:12:52.971298 IP6 2001:0:0:1::15.5683 > aaaa::1.5388: UDP, length 4
17:13:02.019044 IP6 aaaa::1.5388 > 2001:0:0:1::15.5683: UDP, length 17
17:17:27.107296 IP6 aaaa::1.5388 > 2001:0:0:1::15.5683: UDP, length 16
17:17:52.109244 IP6 aaaa::1.5388 > 2001:0:0:1::15.5683: UDP, length 18
17:17:52.110915 IP6 2001:0:0:1::15.5683 > aaaa::1.5388: UDP, length 4
17:18:01.108689 IP6 aaaa::1.5388 > 2001:0:0:1::15.5683: UDP, length 17
17:18:28.139665 IP6 aaaa::1.5388 > 2001:0:0:1::15.5683: UDP, length 16
17:19:23.196270 IP6 aaaa::1.5388 > 2001:0:0:1::15.5683: UDP, length 18
17:19:23.198253 IP6 2001:0:0:1::15.5683 > aaaa::1.5388: UDP, length 4
17:19:29.230863 IP6 aaaa::1.5388 > 2001:0:0:1::15.5683: UDP, length 16
17:20:54.282330 IP6 aaaa::1.5388 > 2001:0:0:1::15.5683: UDP, length 18
17:20:54.283645 IP6 2001:0:0:1::15.5683 > aaaa::1.5388: UDP, length 4
```

Question 6.2.2: 5.0x notifications

Introduce some errors in the `render_post` method, such as a division by 0. What will be the notification code ?

6.2.5 Returning errors

In this scenario, the server will only return `4.0x` and `5.0x` notifications, the simplest rule will be to match for any of these values and just return the RuleID without any residue.

The rule `255/8` given in the file `lpwan.json` (cf. Listing 6.1 on page 86) does not send any residue to the Device. All fields are elided.

1	1	1	1	1	1	1	1

FIGURE 6.2 – IPv6/UDP/CoAP Acknowledgment

A Device receiving that ruleID known that the incoming message was matching the rule. Therefore, it will be notified that an error occurs, but without knowledge of the error.

Question 6.2.3: Positive Notification

Suppose that an old CoAP server does not recognize the *No Response* option. What are the consequences for the Device ? How do we solve this ?

6.2.6 Processing errors on the Device

dev-lpwan.py

Let us go back to the `dev-lpwan.py` program on the Device.

```
    if not wait_ack():
110     event_queue.append((sensor, int(time.time() + \
                                sensor.get_period())))
112 else:
```

```
            print ("Long␣sleep␣for", sensor.uri)
114         event_queue.append((sensor, int(time.time() + 300)))
```

After sending the request, the Device calls the function `wait_ack`. If the result is False, this means that no SCHC packet containing the rule 255/8 has been received. In that case, the next transmission is scheduled in its regular period (lines 110 and 111).

Otherwise, the device will wait 5 minutes before trying to send that particular resource again.

```
48  def wait_ack():
        # wait a SCHC pkt for 1 second
50      readable, _, _ = select.select ([tunnel], [], [], 1)
        if len(readable) == 1: # A message
52          msg = tunnel.recv(1000)
            if msg == b"\xff": # ruleID 255/8 = server error
54              print ("Rule␣255/8:␣Server␣Error")
                return True
56
        return False
```

This function uses the Linux system call `select` to bound in time an answer. If a message is received on the socket used to communication with the core (line 51), then the Rule ID is tested (line 53), to verify that the rule ID corresponds to error messages.

```
$ python3 dev-lpwan.py
sleeping 1 sec.
MID 1 URI temp (index: 0 ) value b'13'
sending: b'0413'
sleeping 28 sec.
MID 2 URI pres (index: 2 ) value b'1903e7'
sending: b'0a1903e7'
Rule 255/8: Server Error
Long sleep for pres
sleeping 10 sec.
MID 3 URI humi (index: 1 ) value b'182d'
sending: b'0d182d'
sleeping 21 sec.
```

We can see that the Device receives a SCHC message in response to the second POST on the `/pres` resource. The device goes into long sleep for this URI.

Question 6.2.4: Handling delays

In this scenario, the notification is expected to arrive 1 second after the transmission of the request. In an LPWAN environment, the downlink message may be delayed to respect a Duty Circle. Does the system continue to work ? How do we solve this problem ?

6.3 Underwater scenario

In the underwater scenario, the Devices are considered as server and will return a value when requested by a boat arriving near-by. The files are in the `MOOC-4` directory. We will use a regular CoAP client based on aiocoap and a constrained server.

6.3.1 The basic rules

Let us start with a single rule for uplink and downlink traffic.

Listing 6.5 – underwater.json

```
{
    "DeviceID" : "udp:10.0.0.20:8888",
    "SoR" : [{
        "RuleIDValue" : 1,
        "RuleIDLength": 3,
        "Compression": [
            {"FID": "IPV6.VER", "TV": 6, "MO": "equal", "CDA": "not-sent"},
            {"FID": "IPV6.TC",  "TV": 0, "MO": "equal", "CDA": "not-sent"},
            {"FID": "IPV6.FL",  "TV": 0, "MO": "ignore","CDA": "not-sent"},
            {"FID": "IPV6.LEN",          "MO": "ignore","CDA": "compute-length"},
            {"FID": "IPV6.NXT", "TV": 17, "MO": "equal", "CDA": "not-sent"},
            {"FID": "IPV6.HOP_LMT", "TV" : 255,"MO": "ignore","CDA": "not-sent"},
            {"FID": "IPV6.DEV_PREFIX","TV": "AAAA::/64",
                                   "MO": "equal","CDA": "not-sent"},
            {"FID": "IPV6.DEV_IID", "TV": "::1","MO": "equal","CDA": "not-sent"},
            {"FID": "IPV6.APP_PREFIX", "TV": "2001:0:0:1::/64",
                                   "MO": "equal","CDA": "not-sent"},
            {"FID": "IPV6.APP_IID", "TV": "::15",
                               "MO": "equal","CDA": "not-sent"},

            {"FID": "UDP.DEV_PORT",  "TV": 5683,"MO": "equal",  "CDA": "not-sent"},
            {"FID": "UDP.APP_PORT",            "MO": "ignore",  "CDA": "value-sent"},
            {"FID": "UDP.LEN",       "TV": 0,   "MO": "ignore", "CDA": "compute-length"},
            {"FID": "UDP.CKSUM",     "TV": 0,   "MO": "ignore", "CDA": "compute-checksum"},

            {"FID": "COAP.VER",  "DI": "BI", "TV": 1,   "MO": "equal","CDA": "not-sent"},
            {"FID": "COAP.TYPE", "DI": "DW", "TV": 0,   "MO": "equal","CDA": "not-sent"},
            {"FID": "COAP.TYPE", "DI": "UP", "TV": 2,   "MO": "equal","CDA": "not-sent"},
            {"FID": "COAP.TKL",  "DI": "BI", "TV": 2,   "MO": "equal","CDA": "not-sent"},
            {"FID": "COAP.CODE", "DI": "DW", "TV": 1,   "MO": "equal","CDA": "not-sent"},
            {"FID": "COAP.CODE", "DI": "UP",            "MO": "ignore","CDA": "value-sent"},
            {"FID": "COAP.MID",  "DI": "BI", "TV": 0,   "MO": "ignore", "CDA": "value-sent"},
            {"FID": "COAP.TOKEN",  "DI": "BI", "TV": 0, "MO": "ignore", "CDA": "value-sent"},
            {"FID": "COAP.Uri-Path", "FP": 1, "DI": "DW",
                           "TV": ["temp", "humi", "pres"],
                           "MO": "match-mapping","CDA": "mapping-sent"},
            {"FID": "COAP.ACCEPT", "DI": "DW", "TV": 30, "MO": "equal", "CDA": "not-sent"},
            {"FID": "COAP.Content-Format", "DI": "UP", "TV": 30, "MO": "equal", "CDA": "not-sent"}
        ]}
    ]
}
```

This rule uses *Directional Indicator* to obtain different residues regarding the direction. *Directional Indicator* As shown in figure 6.3.

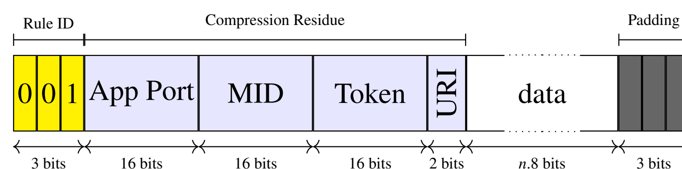


FIGURE 6.3 – IPv6/UDP/CoAP downlink compression

We consider a regular CoAP client based on aoicoap, without any particular settings. The residues are the following :

— The UDP port on 16 bits. If the server port is well-known, the client will select an available port number. There is no way to guess the value that should be sent to the device that will echo it.

— The message ID on 16 bits. As for the UDP port, the Message ID can have any value. aoicoap will select a different value for each transaction, so the Message ID has to be sent integrally to the device.

— The Token value on 16 bits, aiocoap sends by default a token on two bytes. This value is sent to the device. Regarding the scenario, since the device answers immediately, the token is redundant with the message ID.

— The index of requested URI on 2 bits, pointing to the URI.

Since the Rule ID is on 3 bits and the URI index on 2 bits, this compression introduces 3 bits of padding at the end of the SCHC message.

The uplink message has a slightly different format. The device may respond to a GET request with a 2.05 message containing the requested resource or with an error message. Therefore, the CoAP Code must be added to the answer, as shown in Figure 6.4. Note that is the case of an error message, the data are empty.

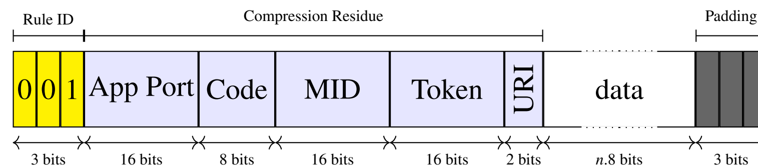


FIGURE 6.4 – IPv6/UDP/CoAP uplink compression

The *Core* uses this rule as usual in the `core.py` program. core.py

6.3.2 The application client

The following program allows the client running on *App* to query for two resources : the `/temp` and the `/humi`. The former returns a value with the 2.05 notification, and the latter returns the 4.04 error notification. These queries are performed every 3 seconds.

Listing 6.6 – coap-client.py

```
import logging
import asyncio
import random

import time

from aiocoap import *

logging.basicConfig(level=logging.INFO)

```

```
async def main():
    protocol = await Context.create_client_context()

    if random.randint(0, 10) % 3 == 0: # wrong URI
        request = Message(code=GET, uri='coap://[aaaa::1]]/humi')
    else: #right URI
         request = Message(code=GET, uri='coap://[aaaa::1]]/temp')

    try:
        response = await protocol.request(request).response
    except Exception as e:
        print('Failed to fetch resource:')
        print(e)
    else:
        print('Result: %s\n%r'%(response.code, response.payload))

if __name__ == "__main__":
    while True:
        asyncio.run(main())
        time.sleep(3)
```

6.3.3 The device server

The server on the device is also quite simple ; we will directly deal with SCHC packets and avoid the decompression.

Listing 6.7 – dev-server.py

```
from re import A
import socket
import binascii
import random
import cbor2 as cbor

import sys

# insert at 1, 0 is the script path (or '' in REPL)
sys.path.insert(1, '../../src/')

import gen_bitarray as ba

tunnel = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
tunnel.bind(('0.0.0.0', 8888))

while True:
    data, addr = tunnel.recvfrom(1500)
    print (binascii.hexlify(data))

    schc_msg = ba.BitBuffer(data)

    ruleID=schc_msg.get_bits(3)

```

```
    if ruleID == 1: # rule 1/3
        app_port = schc_msg.get_bits(16)
        mid = schc_msg.get_bits(16)
        token = schc_msg.get_bits(16)
        uri_idx = schc_msg.get_bits(2)

        uri = ["temp", "humi", "pres", None][uri_idx]

        print(app_port, mid, token, uri_idx, uri)

        schc_resp = ba.BitBuffer()
        schc_resp.add_bits(0b0000_0001, nb_bits=3) # ruleID

        if uri == "temp":
            schc_resp.add_bits(app_port, nb_bits=16) # app port
            schc_resp.add_bits(0b010_00101,nb_bits=8) # 2.05
            schc_resp.add_bits(mid, nb_bits=16) # Message ID
            schc_resp.add_bits(token, nb_bits=16) # Token

            data = cbor.dumps(random.randint(10, 100))
            schc_resp.add_bytes(data)

        else: # 4.04 not found
            schc_resp.add_bits(app_port, nb_bits=16) # app port
            schc_resp.add_bits(0b100_00100,nb_bits=8) # 4.04
            schc_resp.add_bits(mid, nb_bits=16) # Message ID
            schc_resp.add_bits(token, nb_bits=16) # Token

        tunnel.sendto(schc_resp.get_content(), addr)
```

The program opens a socket to communicate with the Core and starts an infinite loop on line 17 to receive requests from the clients.

To facilitate the manipulation of SCHC packets that are bit-aligned, we use openSCHC BitBuffer:

BitBuffer

gen_bitarray

— Line 23 reads the ruleID, since we have a single rule, there is no ambiguity on the size. The method reads 3 bits from the BitBuffer.

r

— Line 24 checks if its the correct rule, then

— Lines 26 to 29 store in variables the different residue imposed by rule 1/3.

— On lines 35 and 36, the SCHC response starts to build by inserting the rule ID into the schc$_r$*espBitBuffer.Line*38*testsiftheURIisknown*(/temp).*TheSCHCresponsewilldifferregardingthe*

— Lines 39 to 45 the posivite notification to the GET request (code 2.05) is added to the response BitBuffer, including the data represented here by a random number converted in CBOR.

— Lines 48 to 51 the negative notificationn (4.04) is added to the BitBuffer.

— Line 53 sends the SCHC message to the Core.

6.3.4 Several Optimization

We defined a generic rule to handle CoAP traffic. In this section, we are going to modify this rule to reduce bandwidth.

Question 6.3.1: 2 rules

The `underwater.json` Set of Rules (cf. page 94) contains a single rule used bidirectionally to compress request and response. Modify this Set of Rules, to handle specifically the notification errors.

What is the impact on traffic ?

7. Fragmentation

Youtube

Fragmentation is a mandatory step for LPWAN networks. The maximum payload (or Maximum Transmission Unit) is about 50 bytes in Europe and 11 in the US. Sigfox MTU is between 8 and 12 bytes. This could be enough to send a value, but some messages are larger and RFC 8200 specifying IPv6 imposes that messages of 1280 bytes must be at least carried by a Layer 2 [1].

In fact, fragmentation is a common thing in networks and can be found at different levels, for instance :

- IP allows to fragment to large UDP messages, natively in IPv4 and through extensions in IPv6,
- 6LoWPAN includes a fragmentation header to transport larger IPv6 packet than allowed by protocols such as IEEE 802.15.4.
- CoAP, through the Blocks options, allows the sender to cut a resource in small part. The requester will have to request individually each part.

None of these behavior are optimal in an LPWAN environment. In IPv6 or 6LoWPAN fragmentation, none of the fragments is acknowledged, so if a fragment is lost, the entire

1. this lower bound is impose to limit the impact of an attack on path MTU, a router can refuse to forward a too large packet and send back by ICMPv6 the maximum size it expects. If very small values are allowed, the performance of the network will be reduced.

packet is lost. On the other hand, in CoAP, each fragment is acknowledged, creating a bidirectional traffic, incompatible with some LPWAN technologies which drastically limit the downlink channel and the extra traffic may also drain the battery faster.

SCHC in RFC 8724 defines three fragmentation modes :

— No acknowledgments (NoAck) : this mode is unidirectional, The sender sends its fragment one by one and adds to the last one a Reassembly Check Sequence (RCS) that allows the receiver to detect if no fragment has been lost. NoAck

— Ack Always (AA) : The message to be fragmented is divided into several windows. Each window must be acknowledged by the receiver to send the next one. The acknowledgment messages contains a bitmap indicating which fragments are missing in the window. As for NoAck, the last fragment contains an RCS to validate the message. AA

— Ack on Error (AoE) : The message is also divided into windows, but the receiver responds by acknowledging only if some fragments are missing in the window. Only the last fragment must be acknowledged to inform the sender that the receiver received the full message. AoE

7.1 Rules

As for compression, rules play an important role in fragmentation. We saw with compression that rules are only shared by two entities. The same is true for fragmentation, but this time they contain fragmentation parameters on which both ends must agree. Usually, compression and fragmentation share the same Rule ID space, so a message which does not need to be fragmented can be directly sent.

If compression rules are designed to be bidirectional, then it is not the case for fragmentation rules for which a direction must be specified. The rule used in the other direction is a synonym of acknowledgment. In LPWAN uplink and downlink do not share the same characteristic, therefore, a different fragmentation algorithm or at least different parameters can be applied in each direction. Figure 7.1 on the facing page illustrates the combination of compression and fragmentation rules.

Figure 7.1 on the next page illustrates the combination of compression and fragmentation rules. A first packet is compressed using rule `1/8` and send directly to the receiver. The second packet is compressed with rule `2/8` but the result has to be fragmented. Rule `3/8` is used to indicate the fragmentation mode and the fragmentation position. The same rule ID is used to acknowledge the sequence.

The receiver replies, and its answer may also be compressed with rule `2/8` and this time fragmented by rule `4/8` which requires no acknowledgment.

In OpenSCHC, a fragmentation rule is defined with the `Fragmentation` key followed by a JSON Object containing at least the fragmentation mode and the fragmentation direction

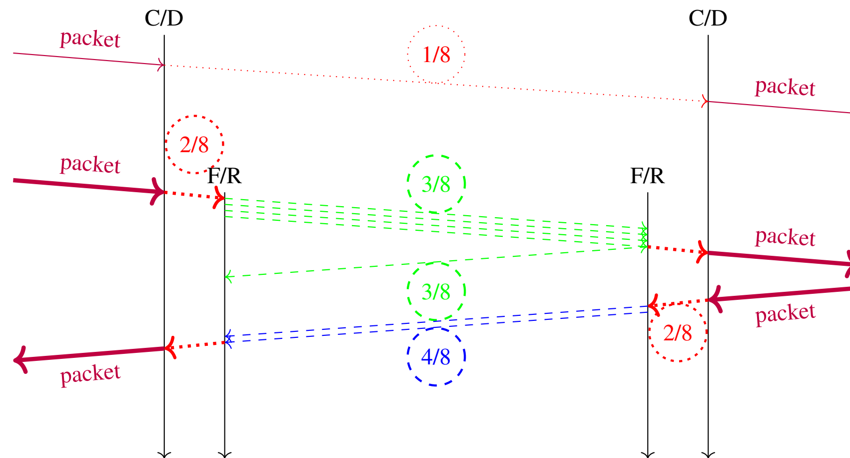


FIGURE 7.1 – Compression and Fragmentation

as shown in the Listing 7.2 on page 105.

Listing 7.1 – Simple fragmentation rule

```
...
        },{
                "RuleID" : 4,
                "RuleIDLength" : 8,
                "Fragmentation" : {
                        "FRMode": "NoAck",
                        "FRDirection": "UP"
                }
        }
```

We will explore various modes of fragmentation, which will assist in presenting progressively all the relevant fields and terminology associated with fragmentation.

7.2 No Ack

There is no acknowledgment in this mode, the sender sends fragments, and the receiver checks if all the fragments are received before delivering the reassembled message to the decompression layer. This leads to two kinds of SCHC fragments :

— the `All-0` [2] fragments presented in Figure 7.2 on the next page is used for all the

2. The name comes from the fact that all bits in the FCN are set to 0, even if there is only one bit in this

fragments but the last. It is composed of the ID of the rule that points to a *No Ack* rule, followed by a Fragment Compressed Number field on 1 bit and set to 0. The rest of the fragment contains a part of the original message. Note that the payload size is in bit and is not aligned to bytes. But in general layer 2 transports bytes, so the SCHC Fragmentation Header and the fragment payload have to be aligned on a byte boundary [3]
For example in Figure 7.2, the rule ID is 1 bytes, the FCN is 1 bit, if the MTU is 12 bytes, 87 bits can be sent.

— The last fragment, called `All-1`, depicted in Figure 7.3differs for the *All-0* in several ways :
  — The FCN value is set to 1,
  — A Reassembly Check Sequence is added at the end of the SCHC header. The size and the algorithm can be specified, but RFC 8724 defines the default value with the Ethernet CRC. The sender takes the original sequence and computes the CRC, the receiver concatenates the received fragments, and performs the same computation. If the result differs from the one sent, then some fragments are discarded or received in the wrong order.
  — padding may be introduced at the end of the last SCHC fragment. Since header compression residues are not bit aligned, as well as rule ID and SCHC fragmentation header, the chance to not reach a byte boundary on the last fragment is high. In any case, the padding may exceed 7 bits. See Section 7.2.1 on the facing page on padding management for more explanations.

padding

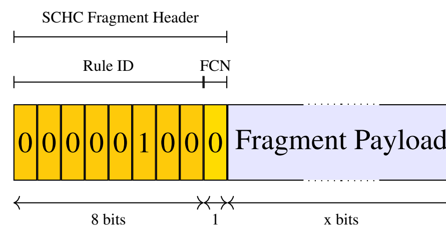


FIGURE 7.2 – All-0 SCHC Fragment in No-Ack mode

In OpenSCHC, the Reassembly Check Sequence (RCS) field corresponds to the result of using the CRC32 algorithm as recommended by RFC 8724, calculated on the full SCHC packet (after reassembly) concatenated with the padding bits. It is possible to specify some other RCS algorithm, for example, in Sigfox, RCS can be the number of fragments in the last window. This limits the fragmentation header size, but weakens the error detection, especially if L4 checksum is removed during the compression phase.

Sigfox

example.

3. To be more generic, RFC 8724 uses L2 Word as a transmission unit. We will use the term byte to simplify.

L2 Word

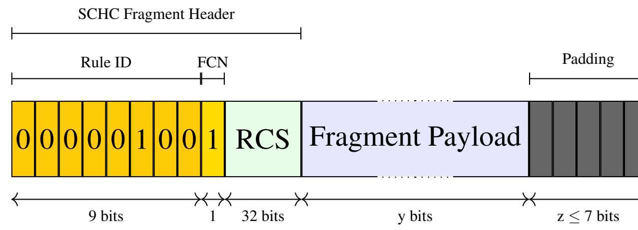


FIGURE 7.3 – All-1 SCHC Fragment in No-Ack mode

7.2.1 Padding management

One nice feature of SCHC is that it avoids carrying the SCHC message length in the header and that it removes padding just through simple mathematical operations.

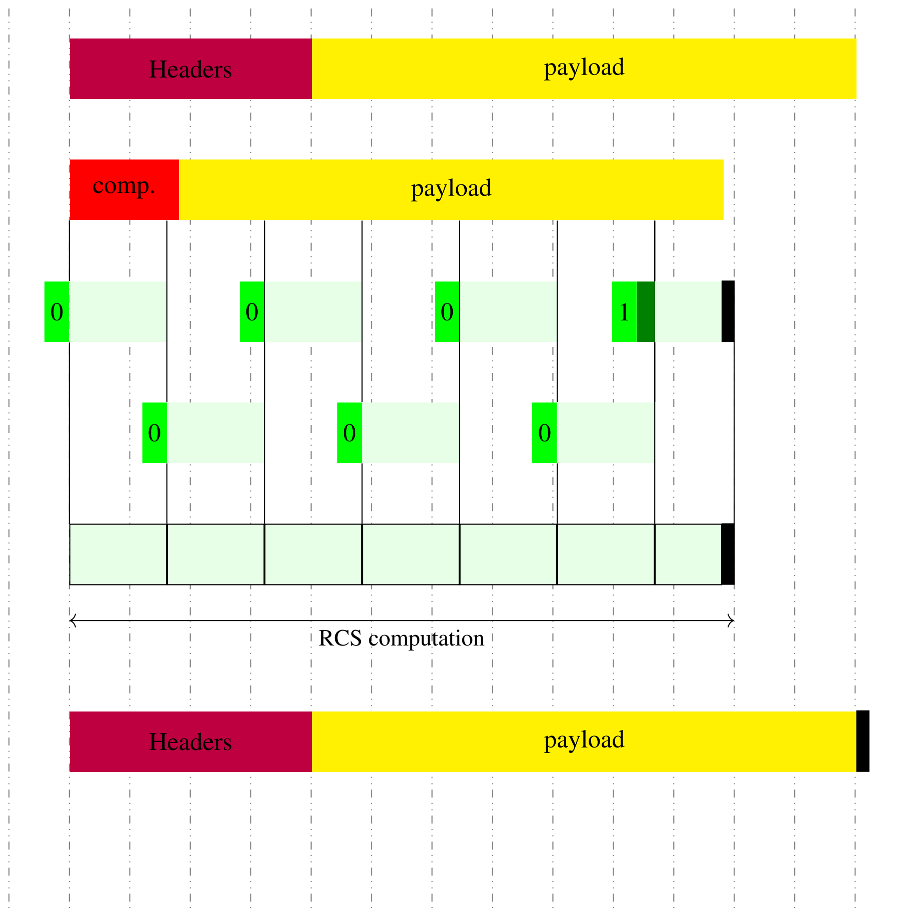


FIGURE 7.4 – Padding management

Figure 7.4 represents the compression / decompression and fragmentation / reassembly processes from the memory perspective and more particularly alignment in bytes (or more precisely on L2-word).

Before compression, the packet is aligned in memory. The header is a multiple of bytes as well as the payload. After compression, the SCHC Header composed of the Rule ID and compression residues may lose this alignment. The payload remains a multiple of bytes.

This SCHC message is fragmented. Each fragment is composed of a SCHC fragmentation header and a payload. The SCHC fragment is a multiple of bytes, but the payload is not aligned on bytes. The last fragment (All-1) may not have enough remaining bits to have a size multiple of bytes, therefore some padding bits (represented in black on the figure must be added).

These padding bits are added to the SCHC message when the receiver proceeds to reassembly. That is why they must be included in the RCS computation [4].

After decompression, the header recovers its alignmen and the padding bits are easily eliminated.

7.2.2 Impact of FCN size

If Layer 2 is changing the packet order, it may be useful to increase the FCN size.

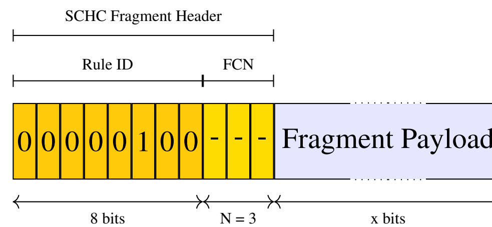


FIGURE 7.5 – All-0 SCHC Fragment in No-Ack mode with larger FCN field

Figure 7.5 shows a *No Ack* packet with an extended FCN, here on 3 bits. FCN numbers are sent in decreasing order. The maximum value where all bits are set to 1 is reserved to indicate the end of the fragmentation. In this example, the FCN size in coded on 3 bits, so the maximum value for FCN is 6 going down to 0. Value 7 as an All-1 indicates the end of the fragmentation and that an RCS follows the header, as shown in Figure 7.6 on the next page.

RFC 8724 defines `M` as the size in bits of the FCN field. The maximun FCN value is therefore $2^M - 2$ and for the All-1 the FCN value is $2^M - 1$. The values from $2^N - 2$ to All-0 or All-1 will define a window [5].

In OpenSCHC, the key `FCNSize` must be added to the fragmentation parameters. By default for NoAck the size is 1 bit.

`FCNSize`

4. We will see later, that in some cases this reassembled message with the padding may not be a multiple of bytes. But most of the CRC library assume that the sequence is a byte string, so some extra zero bits may be added for this purpose.

5. Should the window exceed a manageable size, such as when a device is limited to storing 100 fragments, N is set to 7, and a new rule parameter named `WINDOW_SIZE` as specified in the standard is set to 99.

`WINDOW_SIZE`

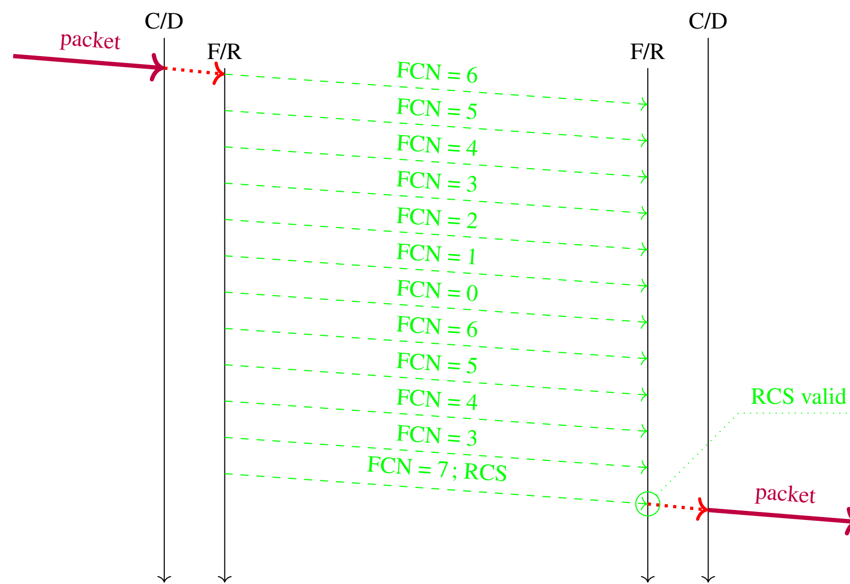


FIGURE 7.6 – NoAck with FCN

Listing 7.2 – Simple fragmentation rule

```
...
        },{
        "RuleID" : 4,
        "RuleIDLength" : 8,
        "Fragmentation" : {
            "FRMode": "NoAck",
            "FRDirection": "UP"
            "FCNSize": 3,
        }
 }
```

7.2.3 Conclusion

To summarize, NoAck is a straightforward mechanism that operates without needing receiver feedback. It is suitable for uplink fragmentation in certain reliable networks. During transmission, if the L2 MTU varies, the fragment will adjust accordingly. However, this mechanism is unsuitable for transmitting critical information, like alarms, especially when there is a high error rate.

Question 7.2.1: FCN size

What will be the maximum FCN value if this field is coded 4 bits ? What will be the All-1 value of the FCN ?

Question 7.2.2: Packet lost

What append if the All-1 fragment is lost ?

7.3 Hands-on No Ack

7.4 Dtag

Occasionally, multiple fragmentation rules are required to manage multiple packets simultaneously. A potential solution is to establish multiple identical fragmentation rules that vary solely by their rule ID. However, this approach complicates the selection of rules. To distinguish between multiple datagrams that utilize the same fragmentation rule, the Datagram Tag (DTAG) was developed. The dimension of this field is determined by the rule implementer [6].

Given that DTAG extends the Rule ID, it directly succeeds it as shown in Figure 7.7.

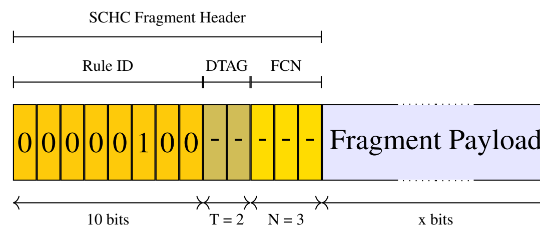


FIGURE 7.7 – DTAG on a NoAck fragment.

7.5 Ack Always

Ack Always is based on a sliding window, which must be acknowledged. In No Ack, we have seen that FCN are decreasing. In fact, the All-0 fragment masks the end of the windows. In Ack Always, windows are numbered on 1 bit and, so have alternatively value 0 and 1.

Upon receiving an All-0 fragment, the receiver is required to confirm the entire window by sending a bitmap that shows which fragments have been received within that window. If this matches the window sent by the sender, the process moves to the subsequent window.

Thus, the format of the fragment header should incorporate a window field of one bit length, as illustrated in Figure 7.8 on the next page.

A fragmentation rule is directional ; hence, the reverse direction is employed to acknowledge the fragment and the structure varies. Two formats for acknowledgment exist : one incorporates a bitmap, and the other, used in the final window, simply confirms that the RCS is correct, indicating successful message reception. The `C` bit distinguish these two formats :

6. noted `T` in RFC 8724.

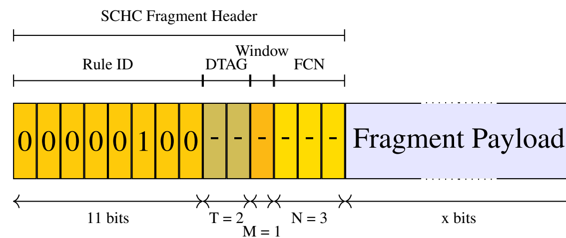


FIGURE 7.8 – Ack Always Fragment Format.

— `C` = 0 a bitmap follows, used in all All-0 windows and when the RCS is wrong on the All-1 windows,

— `C` = 1, only used in the All-1 window, the reception is successful, no need for bitmap.

.

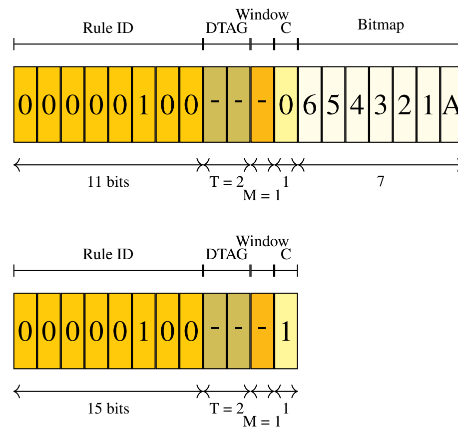


FIGURE 7.9 – Ack Always Ack Formats.

Figure 7.9 gives these two formats when the FCN field is 3 bits long. The numbers represent the acknowledged fragment (`0` the fragment has not been received). The rightmost bit, represented in the figure by an `A` is for All-0 or All-1.

Before sending, the bitmap is also compressed to reduce its size by removing the bytes equal to `0xFF` after the last error or padding bits (not represented in the figure) may be added to fit the byte boundary.

7.5.1 Sliding Window

Figure 7.6 on page 105 depicted a NoAck exchange in which the FCN rolled back to its highest value when it reached All-0. In the Ack Always scenario (refer to Figure 7.10 on the following page), along with a window number, the transmission stops following the emission of the All-0.

This example shows a successful exchange with no losses. Note that in the last window,

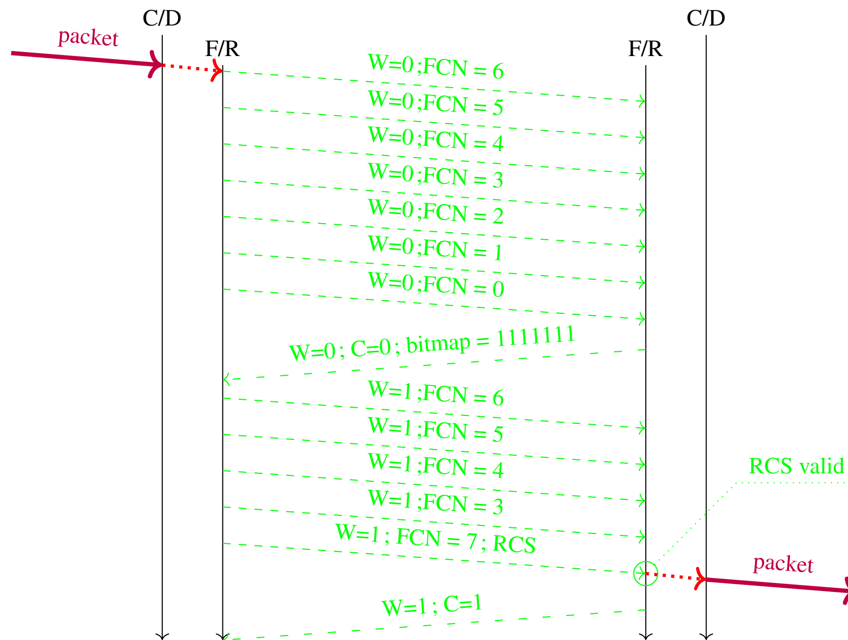


FIGURE 7.10 – NoAck with FCN

the All-1 window is not full, but the RCS allows the receiver to verify that no packet was lost.

7.5.2 Missing fragment

In this subsequent illustration, Figure 7.11 on the next page, there is a loss of some fragments. Consequently, the bitmap shows the absent fragments, prompting the sender to send them again. Observe that the final bitmap is incomplete ; any bit set to 0 signifies either a missing or a non-existent fragment. Should the sender fail to recognize a lost fragment and the RCS proves incorrect, the transmission is deemed compromised, leading to the termination of the fragmentation process.

7.5.3 Bitmap Request

Transmission errors are also possible in All-x fragments, which are more sensitive compared to others due to their ability to elicit a response from the receiver. The loss of the acknowledgment is comparable. The conventional solution involves implementing a timer on the sender's side, which upon expiration prompts the sender to request a bitmap. Rather than creating a new message, SCHC uses an empty All-0 fragment.

In Figure 7.12 on page 110, a SCHC message is divided into four fragments. It fits within a single window, and the final fragment is an All-1. This segment goes missing, leading to no acknowledgment for the sender. Upon expiration of the timer, the sender issues a Bitmap request message, represented as an empty All-0. The receiver attempts to send back the bitmap, but in this example this too is lost. Following another timer expiry, the

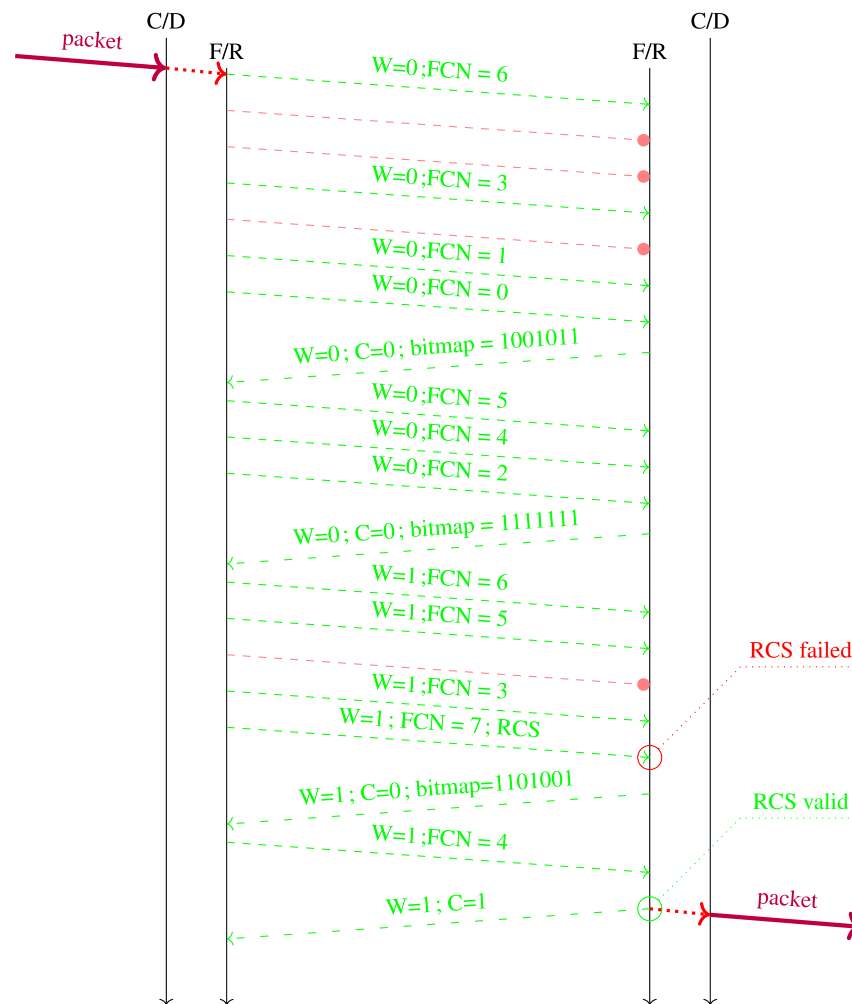


FIGURE 7.11 – NoAck with FCN and packet lost.

sender again requests the bitmap. This time, the bitmap successfully reaches the sender. The absence of a set bit on the right most position of the bitmap signals the loss of the All-1, prompting the sender to resend it. Finally, the RCS is checked out and the receivers issue an acknowledgment, setting the bit `C` to 1.

The timer Retransmission Timer in RFC 8724 is specified in the fragmentation rule parameters and depends on the Layer 2 technology.

7.5.4 Terminating an exchange

If transmission goes well, the exchange will end with an All-1 fragment that carries a C bit equal to 1. But this message can be lost, for the receiver, the transmission is over, but the sender has no confirmation. The sender may continue to request the final bitmap using a Bitmap Request message.

Two parameters may be defined in the rule to terminate an exchange :

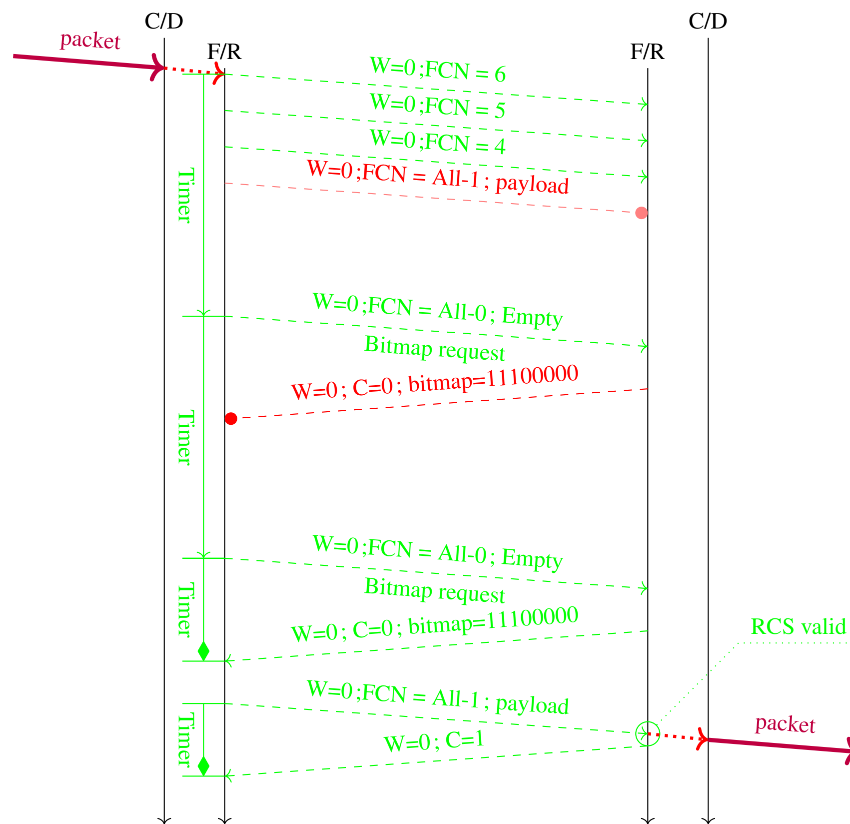


FIGURE 7.12 – NoAck with FCN and All-1 lost

— `Inactivity Timer` defines the maximum time before ending the reception of a message, to protect for instance when the sender is brutally removed from the network during a fragmentation.

— `MAX_ACK_REQUEST` defines on the sender side the maximum bitmap request it can send to recover a window.

Two other messages are specified to inform the other side of the abortion of a fragmentation :

— The sender can send an empty All-1 message.

— The receiver can send an acknowledgment with C set to 1 and at least one byte of bitmap with all bits set to 1.

7.5.5 Byte borrowing

7.5.6 Conclusion

Ack Always is a very reliable mode that allows fragmentation without any limitation in the size of the initial message. The header overhead is small, but it leads to a significant number of exchanges between the sender and the receiver, especially when the acknowledgments are lost.

7.6 Ack on Error

As indicated by its name, Ack on Error diminishes the requirement for Acknowledgment in reporting errors upon detecting losses within a Window, rather than indicating the accurate receipt of an entire window. Ack on error utilizes the same fragment headers as Ack Always, yet expands the Windows field size to encompass the entire fragmented message.

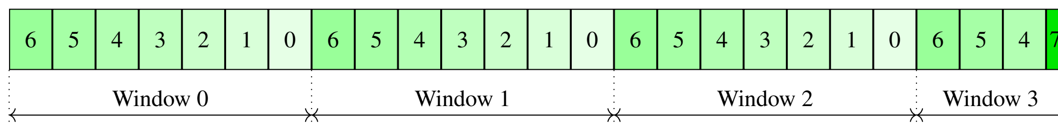


FIGURE 7.13 – Window numbering with Ack On Error

Figure 7.16 on page 114 illustrates the numbering of windows in Ack on Error mode. The FCN value is decremented to the final value of 0, signaling a new window. For our scenario requiring four windows, the Window field in the fragmentation header must have a minimum size of 2 bits.

Question 7.6.1: Sigfox fragmentation

Sigfox network uses an L2 frame with a maximum of 12 bytes. The FCN field size is 4 bits long. The Rule ID is 2 bits long and there is no DTAG. Verify that a window field size of 3 bits is sufficient to transmit a compressed frame of 1280 bytes.

Question 7.6.2: FCN size

With the same hypothesis as in the previous question, what is the window field size if the FCN field size is set to 2 ? and to 5 ?

Since each fragment is uniquely identified with the window and the FCN, this simplifies the protocol. The exchange represented in Figure 7.14 on the following page shows that the bitmap can be send only at the end of the fragmentation when the All-1 is sent.

In this example, the sender sends its three windows. Window 0 is fully received, but windows 1 and 3 are incomplete. The receiver when receiving the All-1 check is that the transmission is complete, all the windows except the last must be full, and the RCS is valid. In our case, some fragments are missing in Window 1, the bitmap is sent, and the sender sends again the missing fragment. Windows 2 is also incomplete, but this can be a normal situation for an All-1 window, but since the RCS fails, the receiver sends the bitmap. The sender detects the missing fragment and sends it. The RCS is validated and the positive acknowledgment with `C` bit set to 1 is sent to the fragmenter to end the session.

Bitmap can be sent at any time, in the scenario we choose to wait the end of the fragmentation. In the standard, it is also possible to send a bitmap after an All-0. But waiting for the end of the fragmentation is a good strategy. In that case RFC 9441 allows the receiver to send several bitmaps in a single message, to limit the number of feedback messages.

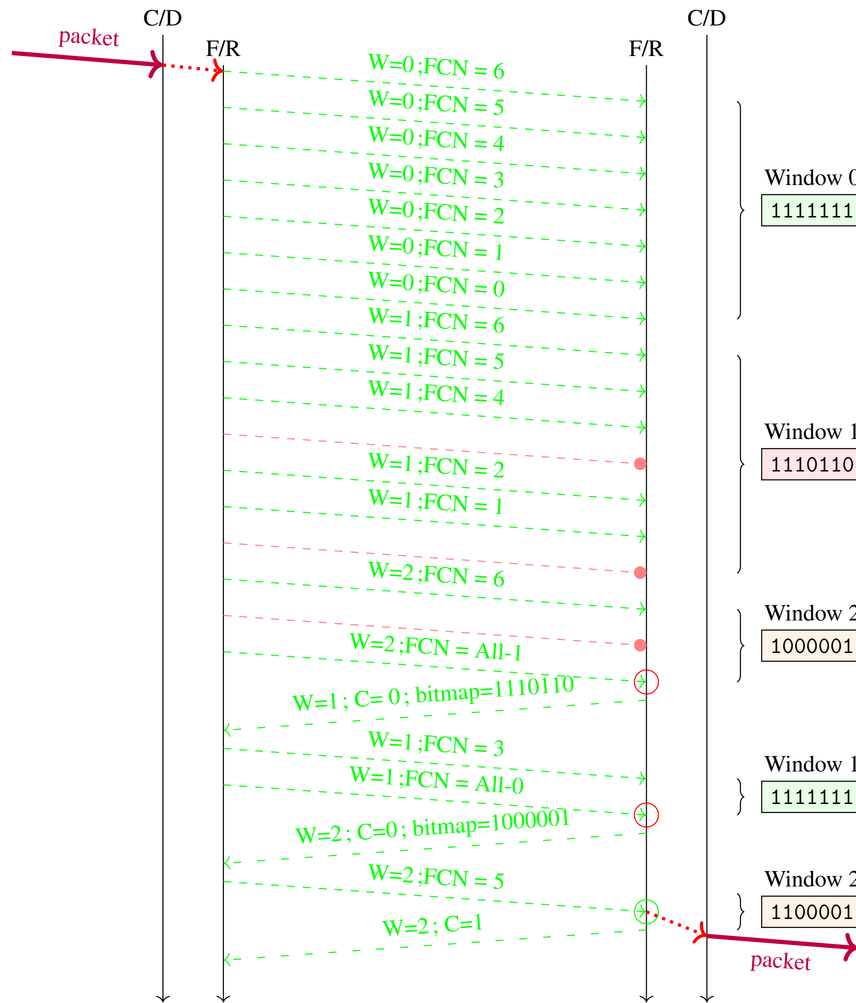


FIGURE 7.14 – Ack on Error with fragment lost

7.6.1 Variable MTU and Tiles

Ack-Always and Ack on Error are unable to dynamically adjust to changes in MTU. For example, in LoRaWAN, moving the spreading factor from SF7 to SF12 decreases the MTU from approximately 200 bytes to 50. Consequently, a fragment transmitted in SF7 cannot be retransmitted if the spreading factor is changed to improve the link quality.

To be insensible to MTU variation, the fragmentation process must be independent of the way the information is sent. Ack on Error introduces the concept of Tile. A tile is a fixed amount of information, chosen to fit the different possible MTU. RFC 9011 specifying SCHC over LoRaWAN has fixed for this technology a Tile on 10 Bytes. The rest of the process is slightly modified. Tiles are numbered in the same way we number fragments with a FCN which should have been renamed in Tile Compressed Number. Then Tiles are physically sent on Fragment, which carries as much as possible consecutive tiles. The fields

Tile

in the fragmentation header contain the window number and the FCN/TCN of the first Tile.

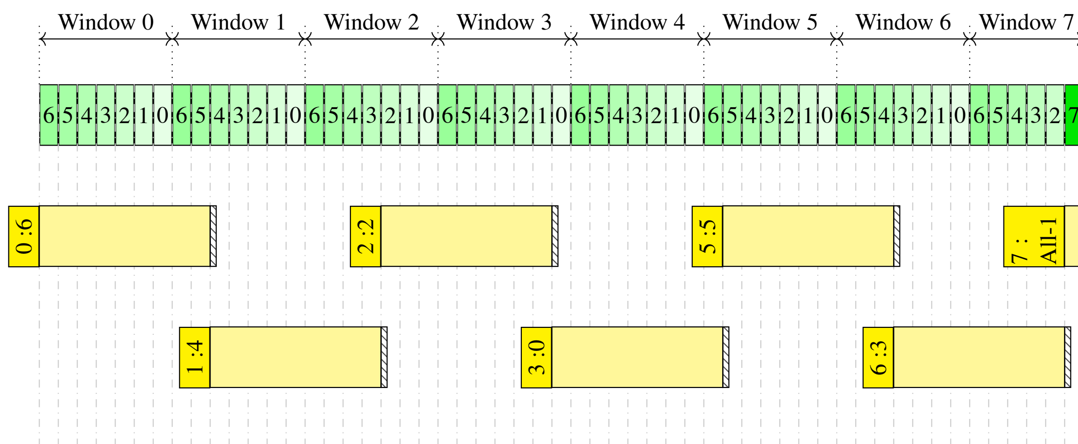


FIGURE 7.15 – Tiles numbering with Ack On Error

In Figure 7.16 on the next page, the initial message is divided into small tiles, leading to more windows than just using the L2 fragment size. The notation `x:y` in the Fragment header indicates the window number (`x`) and the FCN/TCN (`y`). The All-1 Fragment includes also a RCS not represented in the figure.

However, the impact if Tiles on performance is very limited. The window field needs to be 2 or 3 bits larger, as well as the impact on acknowledgment since only incomplete windows are acknowledged. The only constraint is that only the last tile must be in a specific All-1 frame, since All-1 does not give the real position in the original frame. The standard also accepts that the All-1 frame does not contain a tile and must carry the RCS.

Regarding padding, a poor choice of tile length, combined with the length of the fragment header, may result in the addition of bits at the end of the fragment to respect the alignment. This paddig can be easily removed at reception, knowing the fragmentation header size and the tile length. These padding are represented by the hatched pattern on the figure. Only the padding on the All-1 fragment is processed as explained in Section 7.2.1 on page 103.

Fragment lost

Suppose that the Fragment starting with Tile 2 :2 is lost.

This will be represented in Figure 7.17 on the next page as a time diagram.

After receiving the fragments and the All-1 indicating the end of the fragmentation :

— the receiver has windows 2 and 3 completed.

— It sends the bitmap corresponding to window 2, the first incomplete window.

— The sender sends this window, but since the subsequent Tiles were sent in the same fragment, there are also lost, they are also sent in the same fragment.

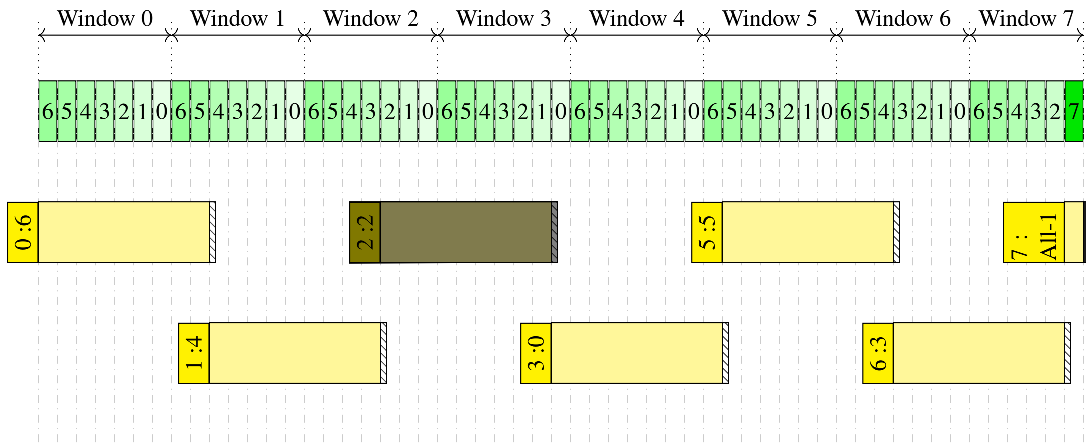


FIGURE 7.16 – Tiles numbering with Ack On Error

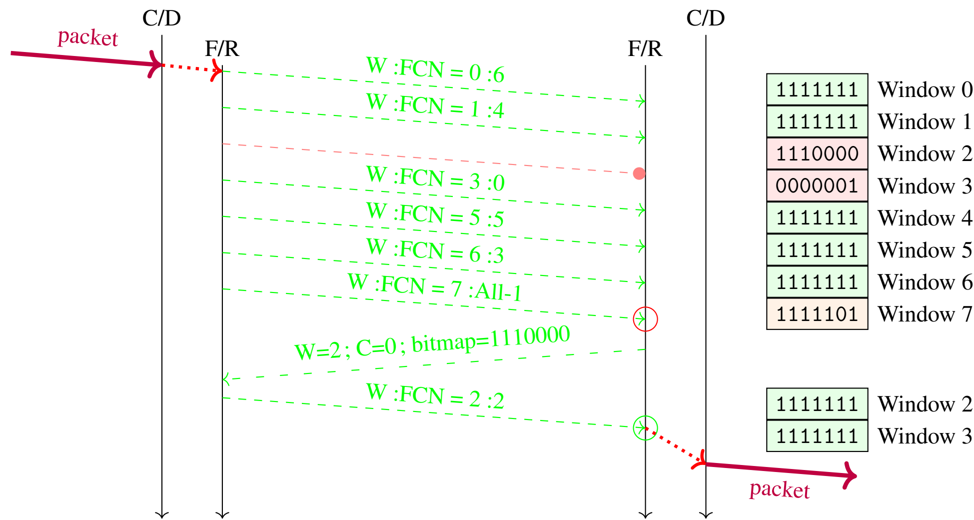


FIGURE 7.17 – Ack on Error with Tiles and fragment lost

— Once the fragment is received, the windows are full and the RCS is valid, so the receiver acknowledges the fragmentation.

Now suppose that during the transmission the MTU has been reduced and only four tiles can now be transmitted. Since the lost fragment contained 9 tiles, this leads to sending three fragments to recover the lost one as shown in Figure 7.18 on the facing page.

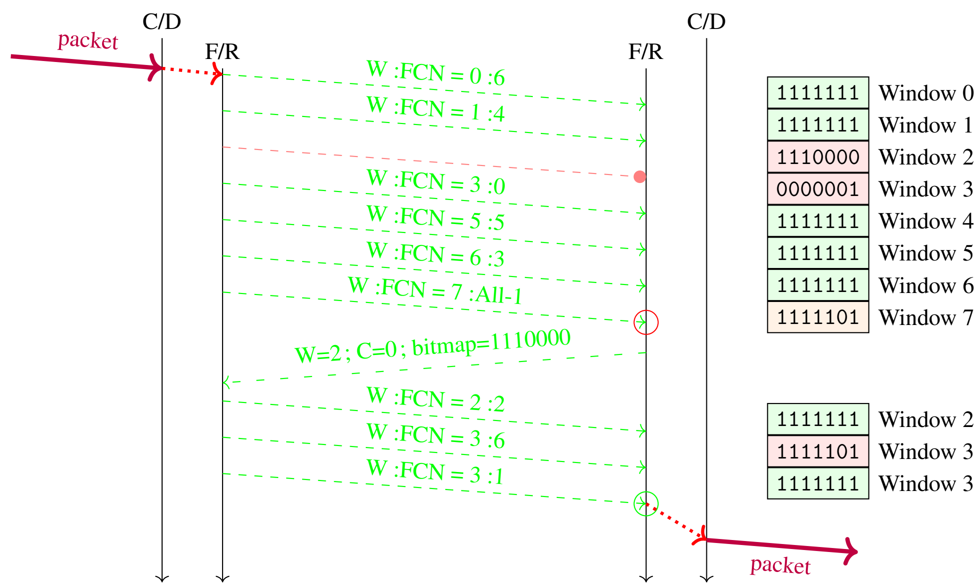


FIGURE 7.18 – Ack on Error with Tiles and fragment lost

8. Ping fragmentation

In this section, we continue exploring the compression of the Ping message. This time we are going to generate larger messages in order to fragment them. The `MOOC-5` directory contains all the files we will need to modify.

8.1 Set up the environment

Launch the virtual machine and the network emulator. For each node (device, core, and app), open a terminal and go to `MOOC-5` directory. Your screen is expected to resemble the one shown in Figure 8.1 on the next page.

On the device terminal, start the ping application `ping_dev1.py`. This program somewhat differs from the one created in chapter 5 on page 68.

Listing 8.1 – ping_dev1.py

```
import sys
# insert at 1, 0 is the script path (or '' in REPL)
sys.path.insert(1, '../../src/')

from scapy.all import *

import gen_rulemanager as RM
from protocol import SCHCProtocol
from gen_parameters import T_POSITION_DEVICE

import netifaces as ni

import socket
import select
import time
```

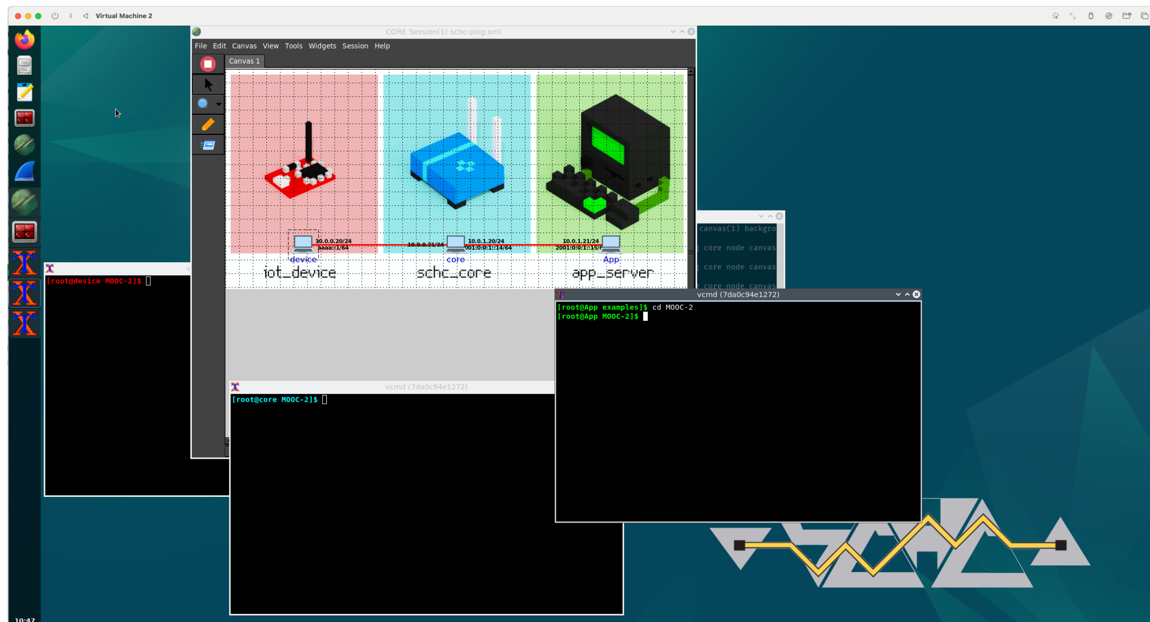


FIGURE 8.1 – Emulation screen for fragmentation.

```

addr = ni.ifaddresses('eth0')[ni.AF_INET][0]['addr']

PORT = 8888
deviceID = "udp:"+addr+":"+str(PORT)

print("device ID is", deviceID)

tunnel = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
tunnel.bind (("0.0.0.0", PORT)) # same port as in the DeviceID
# Create a Rule Manager and upload the rules.

rm = RM.RuleManager()
rm.Add(file="icmp-bi.json")
rm.Print()

core_id = None

# Start SCHC Machine
POSITION = T_POSITION_DEVICE

schc_machine = SCHCProtocol(role=POSITION, tunnel=tunnel)
schc_machine.set_rulemanager(rm)
scheduler = schc_machine.system.get_scheduler()
tunnel = schc_machine.get_tunnel()
```

`AsyncSniffer`

Once the `schc_machine` and its environment have been established, scapy's `AsyncSniffer` is employed. In contrast to `sniff`, calling a function, a thread starts on line 43. It is made

up of an infinite loop.

Listing 8.2 – ping_dev1.py

```
42 t = AsyncSniffer(iface="eth0", store=False)
   t.start()
44 print ("sniff␣started")
   while True:
46     scheduler.run(session=schc_machine)

48     s_in, _, _ = select.select([tunnel], [], [])
       if len(s_in) > 0: # data on the socket
50         SCHC_pkt, device = tunnel.recvfrom(1000)

52         core_id = "udp:"+device[0]+":"+str(device[1])

54         origin, full_packet = schc_machine.schc_recv(
                                   schc_packet=SCHC_pkt,
56                                 device_id=deviceID,
                                   verbose=True)
58
           if full_packet is not None and ICMPv6EchoRequest in full_packet:
60             response =             IPv6Header = IPv6 (
                   version= full_packet[IPv6].version,
62                 tc     = full_packet[IPv6].tc,
                   fl     = full_packet[IPv6].fl,
64                 hlim   = 64,
                   src    = full_packet[IPv6].dst,
66                 dst    = full_packet[IPv6].src
               ) / ICMPv6EchoReply (
68                 id = full_packet[ICMPv6EchoRequest].id,
                   seq = full_packet[ICMPv6EchoRequest].seq,
70                 data = full_packet[ICMPv6EchoRequest].data
               )
72             schc_machine.schc_send(bytes(response),
                                      core_id = core_id,
74                                    verbose=True)
       time.sleep(0.1)
```

The loop will check if some data arrive at the tunnel interface (line 48). If so, the `core_id` is computed from the received message. The fragments are sent to the `schc_machine`. If the fragmentation process is not finished the `schc_recv` returns a `None` value.

If all fragments have been received, the function returns the full packet uncompressed in scapy format.

Line 59 checks if this message is an Echo Request. In that case, it creates an Echo Reply message. This message is returned to the `schc_machine`. `bytes` transforms the scapy message into a byte array and `core_id` tells the `schc_machine` to send it at the core.

Now type :

```
[root@device MOOC-2]$ python3 ping_dev1.py
device ID is udp:10.0.0.20:8888
****************************************
Device: udp:10.0.0.20:8888
/------------------------\
|Rule 6/3          110   |
|--------------+---+--+--+------------------------------+-------------+----------------\
|IPV6.VER      |  4| 1|BI|                            06|EQUAL        |NOT-SENT        |
|IPV6.TC       |  8| 1|BI|                            00|EQUAL        |NOT-SENT        |
|IPV6.FL       | 20| 1|BI|                            00|IGNORE       |NOT-SENT        |
|IPV6.LEN      | 16| 1|BI|------------------------------|IGNORE       |COMPUTE-LENGTH  |
|IPV6.NXT      |  8| 1|BI|                            3a|EQUAL        |NOT-SENT        |
|IPV6.HOP_LMT  |  8| 1|BI|                            ff|IGNORE       |NOT-SENT        |
|IPV6.DEV_PREFIX| 64| 1|BI|              aaaa000000000000|EQUAL        |NOT-SENT        |
|IPV6.DEV_IID  | 64| 1|BI|              0000000000000001|EQUAL        |NOT-SENT        |
|IPV6.APP_PREFIX| 64| 1|BI|------------------------------|IGNORE       |VALUE-SENT      |
|IPV6.APP_IID  | 64| 1|BI|------------------------------|IGNORE       |VALUE-SENT      |
|ICMPV6.TYPE   |  8| 1|DW|                            80|EQUAL        |NOT-SENT        |
|ICMPV6.TYPE   |  8| 1|UP|                            81|EQUAL        |NOT-SENT        |
|ICMPV6.CODE   |  8| 1|BI|                            00|EQUAL        |NOT-SENT        |
|ICMPV6.CKSUM  | 16| 1|BI|                            00|IGNORE       |COMPUTE-CHECKSUM|
|ICMPV6.IDENT  | 16| 1|BI|                            00|IGNORE       |VALUE-SENT      |
|ICMPV6.SEQNO  | 16| 1|BI|                            00|IGNORE       |VALUE-SENT      |
|ICMPV6.PAYLOAD |var| 1|BI|                            00|IGNORE       |VALUE-SENT      |
\--------------+---+--+--+------------------------------+-------------+----------------/
sniff started
```

Similarly, on the core, run `ping_core.py` :

```
[root@core MOOC-2]$ python3 ping_core1.py
****************************************
Device: udp:10.0.0.20:8888
/------------------------\
|Rule 6/3          110   |
|--------------+---+--+--+------------------------------+-------------+----------------\
|IPV6.VER      |  4| 1|BI|                            06|EQUAL        |NOT-SENT        |
|IPV6.TC       |  8| 1|BI|                            00|EQUAL        |NOT-SENT        |
|IPV6.FL       | 20| 1|BI|                            00|IGNORE       |NOT-SENT        |
|IPV6.LEN      | 16| 1|BI|------------------------------|IGNORE       |COMPUTE-LENGTH  |
|IPV6.NXT      |  8| 1|BI|                            3a|EQUAL        |NOT-SENT        |
|IPV6.HOP_LMT  |  8| 1|BI|                            ff|IGNORE       |NOT-SENT        |
|IPV6.DEV_PREFIX| 64| 1|BI|              aaaa000000000000|EQUAL        |NOT-SENT        |
|IPV6.DEV_IID  | 64| 1|BI|              0000000000000001|EQUAL        |NOT-SENT        |
|IPV6.APP_PREFIX| 64| 1|BI|------------------------------|IGNORE       |VALUE-SENT      |
|IPV6.APP_IID  | 64| 1|BI|------------------------------|IGNORE       |VALUE-SENT      |
|ICMPV6.TYPE   |  8| 1|BI|                            80|EQUAL        |NOT-SENT        |
|ICMPV6.CODE   |  8| 1|BI|                            00|EQUAL        |NOT-SENT        |
|ICMPV6.CKSUM  | 16| 1|BI|                            00|IGNORE       |COMPUTE-CHECKSUM|
|ICMPV6.IDENT  | 16| 1|BI|                            00|IGNORE       |VALUE-SENT      |
|ICMPV6.SEQNO  | 16| 1|BI|                            00|IGNORE       |VALUE-SENT      |
|ICMPV6.PAYLOAD |var| 1|BI|                            00|IGNORE       |VALUE-SENT      |
\--------------+---+--+--+------------------------------+-------------+----------------/
```

And finally, on the app, you can send a regular ping to the device.

```
[root@App MOOC-2]$ ping6 aaaa::1
PING aaaa::1(aaaa::1) 56 data bytes
64 bytes from aaaa::1: icmp_seq=1 ttl=255 time=1218 ms
64 bytes from aaaa::1: icmp_seq=2 ttl=255 time=265 ms
64 bytes from aaaa::1: icmp_seq=3 ttl=255 time=1142 ms
64 bytes from aaaa::1: icmp_seq=4 ttl=255 time=213 ms
^C
--- aaaa::1 ping statistics ---
5 packets transmitted, 4 received, 20% packet loss, time 4059ms
rtt min/avg/max/mdev = 212.950/709.578/1217.792/471.549 ms, pipe 2
[root@App MOOC-2]$
```

The device responds and the ping program displays the response. Delays can be quite high because the core SCHC machine is blocked because there is little traffic on the network. We can accelerate the answers by generating traffic in the core. Open a new window on the core and type :

```
[root@core examples]$ ping -i 0.1 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=1.59 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.335 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.425 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.412 ms
...
```

The `-i 0.1` force the ping to send a message every 0.1 seconds. Forget about this window.

Now, if we ping the device, the response time is shorter and more regular.

```
[root@App MOOC-2]$ ping6 aaaa::1
PING aaaa::1(aaaa::1) 56 data bytes
64 bytes from aaaa::1: icmp_seq=1 ttl=255 time=136 ms
64 bytes from aaaa::1: icmp_seq=2 ttl=255 time=237 ms
64 bytes from aaaa::1: icmp_seq=3 ttl=255 time=97.5 ms
64 bytes from aaaa::1: icmp_seq=4 ttl=255 time=93.0 ms
^C
--- aaaa::1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3008ms
rtt min/avg/max/mdev = 93.030/140.875/237.096/58.002 ms
```

8.2 Large pings

We can increase the size of a ping message with the `-s` option [1]. The option `-c 1` forces the ping application to send a single message.

```
[root@App MOOC-2]$ ping6 -s 1000 -c 1 aaaa::1
PING aaaa::1(aaaa::1) 1000 data bytes

--- aaaa::1 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
[root@App MOOC-2]$
```

The core displays this message :

```
make_frag_session, device_id:  udp:10.0.0.20:8888 core_id None direction DW
Protocol fragmentation rule not found
```

The compression result is too large for a single UDP message ; openSCHC limits the size of UDP tunneled messages. You can check that the maximum size for sending without fragmentation is 242 bytes.

8.2.1 NoAck rule

The `icmp-bi.json` file contains a single compression rule, which we used previously to compress ICMPv6 traffic. We need to define a fragmentation rule. Edit the json file to add the following rule :

```
        {
                "RuleID" : 10,
                "RuleIDLength" : 5,
                "Fragmentation" : {
                        "FRMode": "NoAck",
                        "FRDirection": "DW"
                }
       }
```

Rule 10/5 contains minimal information for a NoAck fragmentation mode.

Question 8.2.1: FCN Size

When examining the Rule Manager rules display, how large is the FCN field for rule 4/3 ?

1. Do not increase too much the value. A length of 2 000 bytes will trigger the IPv6 fragmentation extension, which will not be recognized by the openSCHC parser

When receiving the message, the core fragments it into 50 byte long fragment up to the last one which is 6 byte long.

```
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<-- 50--| r:10/5 (noA) DTAG=0 W=- FCN=All-0
<--  6--| r:10/5 (noA) DTAG=0 W=- FCN=All-1
```

At the other side of the UDP tunnel, the Device receives the fragments and reconstruct the original message.

```
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-0|<-- 50--
r:10/5 (noA) DTAG=0 W=- FCN=All-1|<--  6--
SUCCESS: MIC matched. packet bytearray(b'#\xf6\xc91') == result b'#\xf6\xc91'
+-###[ Ethernet ]###
  dst       = 00:00:00:00:00:00
  src       = 00:00:00:00:00:00
  type      = IPv6
###[ IPv6 ]###
     version   = 6
     tc        = 0
     fl        = 0
     plen      = 1008
     nh        = ICMPv6
     hlim      = 255
     src       = 2001:0:0:1::15 [Teredo srv: 0.0.0.1 cli: 255.255.255.234:65535]
     dst       = aaaa::1
###[ ICMPv6 Echo Request ]###
        type      = Echo Request
        code      = 0
        cksum     = 0x8b84
        id        = 0x6877
        seq       = 0x1
        data      = b'\xd0\x15\tg\x00\x00\x00\x00\x80U\n\x00\x00\x00\x00\x00\x10\x11...
        ...
```

Question 8.2.2: Return path

Why the Application do not receive a response to its ping message ?

Question 8.2.3: Reverse Path

Edit the rules on the Device and in the Core to fix the problem.

9. Conclusion

This first edition of the Book of SCHC provides a foundation for understanding and implementing Static Context Header Compression in IoT networks. Through practical examples and hands-on exercises, it demonstrates how SCHC enables efficient communications on constrained networks while maintaining IP compatibility. The combination of theoretical explanation and OpenSCHC implementation offers readers both the knowledge and tools needed to start working with SCHC.

However, SCHC technology and its implementations continue to evolve rapidly. Future editions of this book will significantly expand several important areas of knowledge and practical implementation.

The fragmentation section will be considerably enhanced to cover more advanced topics. Readers will discover detailed explanations of fragmentation reliability mechanisms and learn sophisticated tile management strategies. The management of multiple Maximum Transmission Units (MTUs) will be thoroughly explained, along with specific optimization techniques for different radio technologies. A complete section will be dedicated to the implementation of Request-Response patterns in fragmentation scenarios.

Rule management will see major developments through the integration of CORECONF. This management interface will revolutionize how rules are handled in SCHC implementations. Future editions will explain how to perform dynamic rule updates and distribution, implement proper rule versioning, and manage rule lifecycles. Security considerations in rule distribution will be thoroughly addressed, along with techniques for managing rules across multiple tenants.

A significant addition will come from IMT Atlantique's laboratory, which will provide

real-world embedded implementations. These practical examples will demonstrate SCHC implementation on micro-controllers and show integration with LoRaWAN networks. The reader will learn specific techniques for optimizing battery-operated devices, monitoring real-time performance, and managing memory footprint constraints. These real-world examples will bridge the gap between theory and practical implementation.

The SCHC community continues to grow, and future editions will incorporate field experience from actual deployments. The best practices gathered from the community will provide valuable insights into real-world challenges and solutions. New use cases and applications will demonstrate the versatility of SCHC, while detailed performance optimization techniques will help readers maximize the efficiency of their implementations.

As SCHC-related standards evolve at the IETF, future editions will cover new RFCs and their implementations. The integration of SCHC with emerging IoT standards will be explored, and new compression and fragmentation techniques will be explained as they are developed. Protocol extensions will be thoroughly documented to ensure that readers stay up-to-date with the latest developments.

This book represents a starting point in the journey of SCHC implementation. We encourage readers to follow the SCHC working group at IETF and participate in OpenSCHC development. Sharing implementation experiences and contributing to the growing body of SCHC knowledge will help advance this technology. The foundations provided in this book enable readers to begin implementing SCHC while preparing for future developments in this dynamic field.

Looking ahead, SCHC shows great promise in enabling efficient IoT communications across a wide range of applications and constraints. From smart cities to industrial IoT, from agricultural monitoring to space communications, SCHC will continue to evolve and adapt to new challenges. Future editions of this book will track these developments and provide readers with the knowledge and tools needed to implement increasingly sophisticated SCHC solutions.

SCHOOL

10. Answers to the questions

Question 2.3.1 page 16 Launch `wireshark trace_coap.pcap`. Look at packet 6 in the bottom window. Knowing that the first 14 bytes are the Ethernet header, measure the size of the IPv6 packet.

The network analyzer displays a length of 85 bytes, why does the payload length in the header field contain the value 31 ?

The IPv6 packet is made up of 4 full lines of 16 bytes and 7 additional bytes (2 in the first line and 5 in the last line), which correspond to $4 \times 16 + 7 = 71$ bytes. Payload length fields not include the header length of 40 bytes for IPv6 and the 14 Bytes of Ethernet header.

Question 2.4.1 page 17 To access to a specific field in a packet, scapy uses the following notation `p[L].f`, where `p` is the Scapy frame, `L` is the layer and `f` is the field in that layer. For example, `packet[UDP].sport` gives access to the UDP source port.

Modify the previous program pcap_read.py to display all the Flow Labels used by IPv6.

The following program add all new flow labels in a list.

Listing 10.1 – pcap_flowlabel.py

```
#!/usr/bin/env python3
from scapy.all import *

# rdpcap comes from scapy and loads in our pcap file
packets = rdpcap('trace_coap.pcap')
```

```
flowlabel = []

for packet in packets:
    packet.show()
    if packet[IPv6].fl not in flowlabel:
        flowlabel.append(packet[IPv6].fl)
    hexdump(packet)

    print ("="*40)

print ("Flow Label found:", flowlabel)
```

We have only two labels [479647, 673272] that correspond to both directions of the communication.

Question 2.7.1 page 24 Scapy may not understand CoAP headers and will consider the UDP payload as raw data. Based on `pcap_read.py` write a program that displays the CoAP header and the Token, if exists.

The following program displays the CoAP fields ; `bytes(packet [Raw])` (line 17) transforms the Scapy information into a byte array.

Listing 10.2 – pcap_read_coap.py

```
#!/usr/bin/env python3
from scapy.all import *
import binascii

from torch import binary_cross_entropy_with_logits

# rdpcap comes from scapy and loads in our pcap file
packets = rdpcap('trace_coap.pcap')

type_name = ["CON", "NON", "ACK", "RST" ]

for packet in packets:
    #packet.show()
    #hexdump(packet)
    #print ("="*40)

    coap = bytes(packet[Raw])
    print ("CoAP Header", binascii.hexlify(coap[:4]), end="")
    version = coap[0]>> 6
    type_field = (coap[0] & 0b0011_0000) >> 4
    tkl = coap[0] & 0b0000_1111

    print (": version", version,
           "type", type_field, type_name[type_field],
```

```
25            "token length", tkl, end="")

27      print ("  Code {:3} ({:1}.{:02})".format(
            coap[1],
29          coap[1]>>5,
            coap[1] & 0b000_11111), end="")
31
        print (" MID:", (coap[2]<<8)|coap[3], end="")
33
        if tkl > 0:
35          print(" Token:", binascii.hexlify(coap[4:4+tkl]))
        else:
37          print ("No token")
```

Question 2.7.2 page 24 Using the previous program, or Wireshark, what is the CoAP Port Number used by the server ?

Looking at the nature of the request allows one to determine which host is the server. For example, the first packet in the trace is a GET request, originating from a client to a server ; the destination port gives the port value `5683` for the CoAP server. This is the default value.

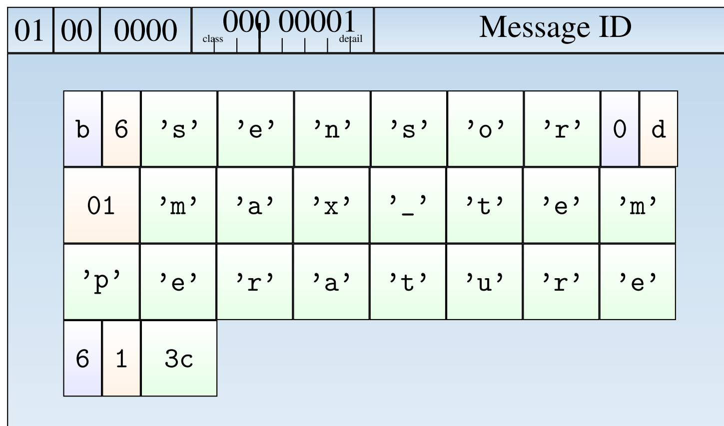
Question 2.7.3 page 25 How do the Message ID evolve between two requests ? Is that mandatory ?

It is incremented. No, the Message ID has to be unique ; it could be chosen randomly, but incrementation is a simple way to guarantee that it will not be reused for another transaction.

Question 2.7.4 page 28 Generate a CoAP Message requesting the content of resource `/sensor/max_temperature` with the CBOR serialization. No token are used.

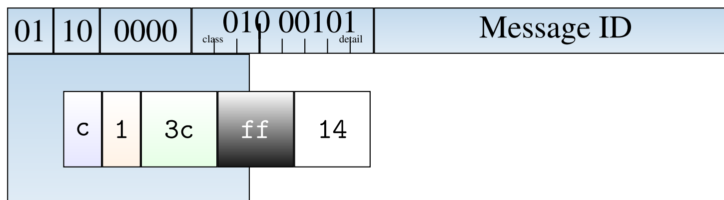
This should be a GET Message `0.01`. The URI path contains two elements, and the second is 14 characters long. The length is coded the following way. The 4 bits contains value `0xd` followed by a byte containing the real value minus 13.

The `Accept` option must be set to 60 (`0x3c`). Since `Accept` option value is 17, it follows `Uri-Path` option with a delta of 6.



Question 2.7.5 page 28 What will be the answer if the maximum temperature is 20°Celsius is immediately returned.

The response uses an `ACK` message and the code will be 2.05. The options indicated the `Content-Format` of 60 indicating a CBOR serialization, followed by the payload marker (`0xFF`) and the value 20 coded in CBOR as `0x14` [1]



Question 2.7.6 page 29 What will be the bitmap value if a client is just interested in `4.xx` and `5.xx` notifications.

Only the `2.xx` should be blocked, so the bitmap is `0000 0010` or `0x02`.

Question 2.8.1 page 32 What is the URI in the previous command ?

`coap://[::1]`

1. See `cbor.me` for convertion.

Question 2.8.2 page 32 Why the request contains no option ?

The URI is implicit, the scheme coap is by default, the authority is the server itself, so it does not need to be added, and the URI path is empty.

Question 2.8.3 page 33 What are the options in the response ?

Etag (4) [2] and Content-Format (4+8=12). The value (0x28) correpond to a link-format, where URI Path (inside <>) sometime with associated properties are listed. Here we have the list of all the URI path corresponding to files and direction in /bin.

Question 2.8.4 page 33 Forge a request, for example coap://[::1]/foo/bar?q=1r=0, and ask for a content in CBOR (value 60). What are the CoAP options ?

- Uri-Path (11) : foo
- Uri-Path (0+11) : bar
- Uri-Query (4+11=15) : q=1
- Uri-Query (0+15) : r=0
- Accept (2+15) : 60 (CBOR)

Question 2.8.5 page 33 What is the goal of the -non option in the aiocoap-client program ?

In that case, NON messages will be used instead of the CON/ACK exchange, the token links requests and responses.

Question 3.4.1 page 42 It is possible to have the same RuleID for a compression rule and a fragmentation rule.

- True

2. Etag is a hash of the content used to see if the value in client's cache correspond to the resource content on the server. Later, the client can insert this value into the If-Not-Match option, the server will return content only if the content has evolved.

— **False**

A ruleID can only designated a single nature of rule.

Question 3.4.2 page 42 Give the binary representation of Rule ID `12/5`

In binary on 5 bits, 12/5 can be writen `01100`

Question 3.4.3 page 42 Which rules are compatible with `12/5` ?

- `3/3`
- `13/5`
- `24/6`
- `26/6`
- `12/6`

The RuleID `3/3` is `011` and starts with the same binary sequence as `12/5`. The RuleID `13/5` is `01101` and is correct, since the binary sequence does not match. The RuleID `24/6` is `011000` and the first 5 bits are similar. The RuleID `24/6` is `001100` and is correct, since the binary sequence does not match.

Question 4.1.1 page 46 Considering the packet above, in hexadecimal and disassembled.

What is the value of the `IPV6.VER` field ?

- `4`
- `40`
- `6`
- `60`

The direction is down-link, what is the value of the `IPV6.DEV_PREFIX` field ?

- `20 01 41 D0 03 02 22 00`
- `00 00 00 00 00 00 13 B3`

- `20 01 41 D0 04 04 02 00`
- `00 00 00 00 00 00 3A 86`

Question 4.1.2 page 47 What are the bits tested for a MSB of 12 bits and a Target Value of 23628.

- `111000100110`
- `101110001001`
- `100110100100`
- `010001011100`

23628 is 0x5C4C in hexadecimal, the 12 first bits are 0x5C4 or 010111000100 in binary.

Question 4.1.3 page 47 The traffic issued from the device and the application may both use IPv6 global and link-local addresses. What will be the Target Value to describe these device prefixes ?

The Target Value will be an array of 2 elements `["2001:41D0:0302: 2200::/64", "FE80::/64"]`.

Question 4.2.1 page 51 Considering the rule given listing 4.2 on page 50. What is the length of the residue ? What is the size of the SCHC packet (assuming that the data part is 40 byte long) ?

In this rule, most fields have the CDA `not sent`. The fields :

- `IPV6.HOP_LMT` is sent on 8 bits,
- both prefixes are compressed with a Mapping List of 2 entries, so the residue will be 1 bit per prefix,
- the `UDP.DEV_PORT` is fully sent on 16 bits and
- the 4 LSB of `UDP.APP_PORT`.

the residue will be $8+2*1+16+4 = 30$ bits long.

To compose a SCHC packet, the Rule ID is added before the residue (3 bits), then the 40 bytes of data and a padding of 7 bits to align the SCHC packet on a byte boundary. Leading to a SCHC packet of 44 bytes.

Question 4.2.2 page 51 Does the following rule match the packet given page 45 matches the rule given in listing 4.2 on page 50.

No, in the packet, the Flow Label contains `673272` and the rule expects `144470`. The rule will be rejected.

Question 4.2.3 page 51 Decompress the following up-link SCHC packet `a4 6e b1 7a a4 34 90 86 85 00`[a].

a. To help you, this is the binary equivalent : `101 00100011 0 1 1101011000101111 0101 01001000 01101001 00100001 00001101 00001010 0000000`

The rule ID is 101 (5/3) followed by Residue :

- version is stored in the rule : 6
- Traffic Class is store in the rule : 1
- Flow Label is stored in the rule : 144470
- Length will be computed when the full packet will be reconstructed.
- Next Header is found in the rule : 17
- Hop Limit is taken from the residue : `00100011` or 35
- Device prefix is taken from the residue (0) pointing to the first entry in the mapping list : `FE80::/64`
- Device IID is stored in the rule `::13b3`, since it is a up-link packet. This will form the source address in the IPv6 packet `FE80::13b3/64`.
- The application prefix is taken from residue (1) that points to the second entry in the mapping list : `FE80::/64`.
- The Application Interface IDentifier (IID) is stored in the rule `2`, since it is a up-link packet. This will form the destination address in the IPv6 packet `FE80::2/64`.
- The Device port is sent in the residue `1101011000101111` or 21447. This will be the source port.
- The Application port is a combination of the 12 first bits of the Target Value 5680 or `0001 0110 0011` combined with the residue of `0101` forming the application port of 5685.
- The UDP length and checksum will be computed when the full packet is reconstructed.

Question 4.3.1 page 56 Adapt the rule stored in `ipv6.json` file (see Listing 4.2 on page 50) to allow for at least one downlink packet matching.

Question 4.3.2 page 57 How to make that rule bidirectional ? You can add a Direction Indicator (key "DI" in openSCHC format, and a value "UP" or "DW") to force the matching in a specific direction.

IPV6.FL

The IPv6 addresses or port numbers are not sensitive to direction, in the example, the Flow Label is different in both directions. So, we need to specify a different TV for each direction. The Listing 10.3 gives the first lines of the rule with the double definition of IPV6.FL.

Listing 10.3 – ipv6-sol-bi.json

```
[{
    "RuleIDValue" : 5,
    "RuleIDLength": 3,
    "Compression": [
      {"FID": "IPV6.VER",
       "TV": 6, "MO": "equal",
       "CDA": "not-sent"},
      {"FID": "IPV6.TC",
       "TV": 0, "MO": "equal",
       "CDA": "not-sent"},
      {"FID": "IPV6.FL", "DI": "DW",
       "TV": 673272, "MO": "equal",
       "CDA": "not-sent"},
      {"FID": "IPV6.FL", "DI": "UP",
       "TV": 479647, "MO": "equal",
       "CDA": "not-sent"},
      {"FID": "IPV6.LEN",
```

This leads to the following full rule.

Listing 10.4 – ipv6-bi

```
****************************************
Device: None
/-------------------------\
|Rule 5/3            101  |
|---------------+---+--+--+------------------------------+-------------+----------------\
|IPV6.VER       |  4| 1|BI|                            06|EQUAL        |NOT-SENT        |
|IPV6.TC        |  8| 1|BI|                            00|EQUAL        |NOT-SENT        |
|IPV6.FL        | 20| 1|DW|                        0a45f8|EQUAL        |NOT-SENT        |
|IPV6.FL        | 20| 1|UP|                        07519f|EQUAL        |NOT-SENT        |
|IPV6.LEN       | 16| 1|BI|------------------------------|IGNORE       |COMPUTE-LENGTH  |
|IPV6.NXT       |  8| 1|BI|                            11|EQUAL        |NOT-SENT        |
|IPV6.HOP_LMT   |  8| 1|BI|------------------------------|IGNORE       |VALUE-SENT      |
|IPV6.DEV_PREFIX| 64| 1|BI|              fe80000000000000|MATCH-MAPPING|MAPPING-SENT    |
:       .       : . :. :. :              200141d003022200:      .      :       .        :
:       .       : . :. :. :              200141d004040200:      .      :       .        :
|IPV6.DEV_IID   | 64| 1|BI|              0000000000003a86|EQUAL        |NOT-SENT        |
|IPV6.APP_PREFIX| 64| 1|BI|              200141d004040200|MATCH-MAPPING|MAPPING-SENT    |
:       .       : . :. :. :              200141d003022200:      .      :       .        :
:       .       : . :. :. :              fe80000000000000:      .      :       .        :
|IPV6.APP_IID   | 64| 1|BI|              00000000000013b3|EQUAL        |NOT-SENT        |
|UDP.DEV_PORT   | 16| 1|BI|------------------------------|IGNORE       |VALUE-SENT      |
|UDP.APP_PORT   | 16| 1|BI|                          1630|MSB(12)      |LSB             |
|UDP.LEN        | 16| 1|BI|                            00|IGNORE       |COMPUTE-LENGTH  |
|UDP.CKSUM      | 16| 1|BI|                            00|IGNORE       |COMPUTE-CHECKSUM|
\---------------+---+--+--+------------------------------+-------------+----------------/
```

Question 4.4.1 page 59 Identify in the trace 10.5, the different elements that make up a SCHC packet (rule ID, residues, data, and padding). You can add the `verbose=True` argument when calling the compresson method.

The verbose option allows us to see how the bitbuffer is filled. The dashed line underneath represents the bits really set.

Listing 10.5 – trace-bitbuffer

```
10100000/3
---
10100000/3
---
10100000/3
---
10100000/3
---
10100000/3
---
10100000/3
---
1010100000000000/11
-----------
1010100000010000/13
-------------
1010100000010000/13
-------------
1010100000010010/15
---------------
1010100000010010/15
---------------
10101000000100110000001101110010/31
-------------------------------
101010000001001100000011011100100110000 0/35
-----------------------------------
10101000000100110000001101110010011000000/35
-----------------------------------
10101000000100110000001101110010011000000/35
-----------------------------------
SCHC packet in hex
a81303726c48b3df07d8bfe646064665a60685a606c4062607462600/219
SCHC packet in binary
1010100000010011000000110111001001101100010010001011001111101
1111000000111110110001011111111100110010001100000011001000110
0110010110100110000000110100001011010011000000110110001000000
011000100110000000111010001100010011000000000/219
```

.

— Line 1, the bitbuffer contains the 3 bits of Rule ID 5/3. Up to line 12, the bitbuffer is not modified ; this corresponds to the first 6 fields of the rule (`IPV6.VER`, `IPV6.TC`,

IPV6.FL, IPV6.LEN, IPV6.NXT) with a not-sent or compute-* CDA.

— line 13, the bitbuffer is increased by 8 bits 01000000 corresponding to the IPV6.HOP_LMT field with a value-sent CDA.

— Line 15 and 17, the two added bits correspond to the index of the Device prefix and the Application prefix.

— Line 23, the 16 bits of the Device UDP port are added, ine 25, the 4 LSB bits are added.

These 35 bits compose the RuleID and the residue part. Line 35, the full SCHC packet includes the payload part and the padding (non-dashed bits).

Question 4.4.2 page 62 In this example, what is the compression ratio of the IPv6/UDP header ?

Original IPv6/UDP header is 40+8 byte long or 384 bits. SCHC compresses it into 35 bits. The header is roughly divided by 10.

Question 4.5.1 page 63 Modify the previous rule to handle the Flow Label and Hop Limit fields as described in this section. Is the checksum modified ?

The rule is as follows :

Listing 10.6 – ipv6-sol-bi-fl

```
Device: None
/-------------------------\
|Rule 5/3            101  |
|---------------+---+--+--+------------------------------+-------------+----------------\
|IPV6.VER       |  4| 1|BI|                            06|EQUAL        |NOT-SENT        |
|IPV6.TC        |  8| 1|BI|                            00|IGNORE       |NOT-SENT        |
|IPV6.FL        | 20| 1|BI|                            00|IGNORE       |NOT-SENT        |
|IPV6.LEN       | 16| 1|BI|------------------------------|IGNORE       |COMPUTE-LENGTH  |
|IPV6.NXT       |  8| 1|BI|                            11|EQUAL        |NOT-SENT        |
|IPV6.HOP_LMT   |  8| 1|UP|                            01|IGNORE       |NOT-SENT        |
|IPV6.HOP_LMT   |  8| 1|DW|                            32|IGNORE       |NOT-SENT        |
|IPV6.DEV_PREFIX| 64| 1|BI|              fe80000000000000|MATCH-MAPPING|MAPPING-SENT    |
:       .       :  . :. :. :              200141d003022200:      .      :       .        :
:       .       :  . :. :. :              200141d004040200:      .      :       .        :
|IPV6.DEV_IID   | 64| 1|BI|              0000000000003a86|EQUAL        |NOT-SENT        |
|IPV6.APP_PREFIX| 64| 1|BI|              200141d004040200|MATCH-MAPPING|MAPPING-SENT    |
:       .       :  . :. :. :              200141d003022200:      .      :       .        :
:       .       :  . :. :. :              fe80000000000000:      .      :       .        :
|IPV6.APP_IID   | 64| 1|BI|              00000000000013b3|EQUAL        |NOT-SENT        |
|UDP.DEV_PORT   | 16| 1|BI|------------------------------|IGNORE       |VALUE-SENT      |
|UDP.APP_PORT   | 16| 1|BI|                          1630|MSB(12)      |LSB             |
|UDP.LEN        | 16| 1|BI|                            00|IGNORE       |COMPUTE-LENGTH  |
|UDP.CKSUM      | 16| 1|BI|                            00|IGNORE       |COMPUTE-CHECKSUM|
\---------------+---+--+--+------------------------------+-------------+----------------/
```

Notice that the conversion for the flow label field is bidirectional and different for the hop-limit field regarding the direction.

The comparison between the original packet and the decompressed one is the following :

```
600a45f8001f1140200141d00302220000000000000013b3200141d00404020000000000000003a86163381b9001f5966...
60000000001f1132200141d00302220000000000000013b3200141d00404020000000000000003a86163381b9001f5966...
```

The checksum is not modified since neither the flow label nor the hop limit is included in its computation.

Question 4.8.1 page 65 How many type of CoAP transactions can you identify ?

Two type transactions :

- a GET to coap ://user.ackl.io/time include in a CONfirmable CoAP message. The positive notification `2.05` is included in the CoAP ACK message that contains the requested resource.
- a PUT to coap ://user.ackl.io/other/block in a CONfirmable CoAP message containing information. The positive notification `2.04` in a CoAP ACK message contains no data.

Question 4.8.2 page 65 What is the size and value of the token ?

The token is 2 byte long.

Question 4.8.3 page 66 What are the code values, are they related to a direction ?

Possible values are GET and PUT for uplink messages and 2.05 and 2.04 for downlink. 2.05

Question 4.8.4 page 66 What are the options for the different types of messages.

- The GET request contains an Uri-host and an a single Uri-path options. Uri-host
- The PUT request contains *Uri-host* and two *Uri-path* options. *Uri-path*
- The answers 2.05 or 2.04 have no options. 2.04

Question 4.8.5 page 66 Define the set of rules (with RuleID 5/3 and 6/3), to compress both IPv6/UDP/CoAP sessions.

Listing 10.7 – rule_coap1

```
****************************************
Device: None
/-------------------------\
|Rule 5/3            101  |
|---------------+---+--+--+------------------------------+-------------+----------------\
|IPV6.VER       |  4| 1|BI|                            06|EQUAL        |NOT-SENT        |
|IPV6.TC        |  8| 1|BI|                            00|IGNORE       |NOT-SENT        |
|IPV6.FL        | 20| 1|BI|                            00|IGNORE       |NOT-SENT        |
|IPV6.LEN       | 16| 1|BI|------------------------------|IGNORE       |COMPUTE-LENGTH  |
|IPV6.NXT       |  8| 1|BI|                            11|EQUAL        |NOT-SENT        |
|IPV6.HOP_LMT   |  8| 1|UP|                            01|IGNORE       |NOT-SENT        |
|IPV6.HOP_LMT   |  8| 1|DW|                            32|IGNORE       |NOT-SENT        |
|IPV6.DEV_PREFIX| 64| 1|BI|              fe80000000000000|MATCH-MAPPING|MAPPING-SENT    |
:       .       :  . :. :. :              200141d003022200:      .      :       .        :
:       .       :  . :. :. :              200141d004040200:      .      :       .        :
|IPV6.DEV_IID   | 64| 1|BI|              0000000000003a86|EQUAL        |NOT-SENT        |
|IPV6.APP_PREFIX| 64| 1|BI|              200141d004040200|MATCH-MAPPING|MAPPING-SENT    |
:       .       :  . :. :. :              200141d003022200:      .      :       .        :
:       .       :  . :. :. :              fe80000000000000:      .      :       .        :
|IPV6.APP_IID   | 64| 1|BI|              00000000000013b3|EQUAL        |NOT-SENT        |
|UDP.DEV_PORT   | 16| 1|BI|------------------------------|IGNORE       |VALUE-SENT      |
|UDP.APP_PORT   | 16| 1|BI|                          1630|MSB(12)      |LSB             |
|UDP.LEN        | 16| 1|BI|                            00|IGNORE       |COMPUTE-LENGTH  |
|UDP.CKSUM      | 16| 1|BI|                            00|IGNORE       |COMPUTE-CHECKSUM|
|COAP.VER       |  2| 1|BI|                            01|EQUAL        |NOT-SENT        |
|COAP.TYPE      |  2| 1|UP|                            00|EQUAL        |NOT-SENT        |
|COAP.TYPE      |  2| 1|DW|                            02|EQUAL        |NOT-SENT        |
|COAP.TKL       |  4| 1|BI|                            02|EQUAL        |NOT-SENT        |
|COAP.CODE      |  8| 1|UP|                            01|EQUAL        |NOT-SENT        |
|COAP.CODE      |  8| 1|DW|                            45|EQUAL        |NOT-SENT        |
|COAP.MID       | 16| 1|BI|------------------------------|IGNORE       |VALUE-SENT      |
|COAP.TOKEN     |tkl| 1|BI|------------------------------|IGNORE       |VALUE-SENT      |
|COAP.URI-HOST  |var| 1|UP|        757365722e61636b6c2e696f|EQUAL        |NOT-SENT        |
|COAP.URI-PATH  |var| 1|UP|                    74696d65|EQUAL        |NOT-SENT        |
\---------------+---+--+--+------------------------------+-------------+----------------/
/-------------------------\
|Rule 6/3            110  |
|---------------+---+--+--+------------------------------+-------------+----------------\
|IPV6.VER       |  4| 1|BI|                            06|EQUAL        |NOT-SENT        |
|IPV6.TC        |  8| 1|BI|                            00|IGNORE       |NOT-SENT        |
|IPV6.FL        | 20| 1|BI|                            00|IGNORE       |NOT-SENT        |
|IPV6.LEN       | 16| 1|BI|------------------------------|IGNORE       |COMPUTE-LENGTH  |
|IPV6.NXT       |  8| 1|BI|                            11|EQUAL        |NOT-SENT        |
|IPV6.HOP_LMT   |  8| 1|UP|                            01|IGNORE       |NOT-SENT        |
|IPV6.HOP_LMT   |  8| 1|DW|                            32|IGNORE       |NOT-SENT        |
|IPV6.DEV_PREFIX| 64| 1|BI|              fe80000000000000|MATCH-MAPPING|MAPPING-SENT    |
:       .       :  . :. :. :              200141d003022200:      .      :       .        :
:       .       :  . :. :. :              200141d004040200:      .      :       .        :
|IPV6.DEV_IID   | 64| 1|BI|              0000000000003a86|EQUAL        |NOT-SENT        |
|IPV6.APP_PREFIX| 64| 1|BI|              200141d004040200|MATCH-MAPPING|MAPPING-SENT    |
:       .       :  . :. :. :              200141d003022200:      .      :       .        :
:       .       :  . :. :. :              fe80000000000000:      .      :       .        :
|IPV6.APP_IID   | 64| 1|BI|              00000000000013b3|EQUAL        |NOT-SENT        |
|UDP.DEV_PORT   | 16| 1|BI|------------------------------|IGNORE       |VALUE-SENT      |
|UDP.APP_PORT   | 16| 1|BI|                          1630|MSB(12)      |LSB             |
|UDP.LEN        | 16| 1|BI|                            00|IGNORE       |COMPUTE-LENGTH  |
|UDP.CKSUM      | 16| 1|BI|                            00|IGNORE       |COMPUTE-CHECKSUM|
|COAP.VER       |  2| 1|BI|                            01|EQUAL        |NOT-SENT        |
|COAP.TYPE      |  2| 1|UP|                            00|EQUAL        |NOT-SENT        |
|COAP.TYPE      |  2| 1|DW|                            02|EQUAL        |NOT-SENT        |
|COAP.TKL       |  4| 1|BI|                            02|EQUAL        |NOT-SENT        |
|COAP.CODE      |  8| 1|UP|                            03|EQUAL        |NOT-SENT        |
|COAP.CODE      |  8| 1|DW|                            44|EQUAL        |NOT-SENT        |
|COAP.MID       | 16| 1|BI|------------------------------|IGNORE       |VALUE-SENT      |
|COAP.TOKEN     |tkl| 1|BI|------------------------------|IGNORE       |VALUE-SENT      |
|COAP.URI-HOST  |var| 1|UP|        757365722e61636b6c2e696f|EQUAL        |NOT-SENT        |
|COAP.URI-PATH  |var| 1|UP|                  6f74686572|EQUAL        |NOT-SENT        |
|COAP.URI-PATH  |var| 2|UP|                    626c6f636b|EQUAL        |NOT-SENT        |
\---------------+---+--+--+------------------------------+-------------+----------------/
```

Question 4.8.6 page 66 In compress rule, identify the different elements that make up

the compression residue.

The SCHC header for GET requests, given the listing 10.8, has been edited to show the composition of the residue.

Listing 10.8 – residue1

```
Original packet size (in byte) 72
Compressed packet (in byte) 8
101 10 01 1000000110111001 0011 1001111011101010 0011111010110111 00000/59
=== == == ================ ==== ================ ================
```

where :

- 101 is the Rule ID,
- 10 is the index for the Device prefix,
- 01 is the index for the Application prefix,
- 1000000110111001 contains the device UDP port
- 0011 are the LSB of the Application UDP port
- 1001111011101010 is the Message ID field
- 0011111010110111 is the token

There are no data following the headers. As we saw, the remaining bits, set to 0, are the padding bits.

The response to the GET is given Listing 10.9.

Listing 10.9 – residue2

```
101 10 01 1000000110111001 0011 1001111011101010 0011111010110111 00110...1100000011100000000/187
=== == == ================ ==== ================ ================ -----...--------------
```

The residue format is the same, followed by the payload.

Question 4.8.7 page 66 Do these rules (see correction on the preceding page) introduce padding bits ?

The compression introduces a 5 bit padding. The not byte-aligned compression CDA is the mapping index for device and application addresses introducing 4 bits and the `MSB/LSB` introducing 4 bits. But since the result is 8 bits, no padding is needed. In fact, the padding comes from the rule ID length of 3 bits, leading to 5 bits padding.

Question 4.8.8 page 66 Propose some solutions to reduce the impact of the padding.

One easy solution is to increase the rule ID length. The padding bits are always lost bits. Increasing the length of the rule allows the creation of more rules.

Question 5.3.1 page 72 What is the nature of these two ICMPv6 messages ?

The first message of type `0x88` (136) is The second message of type `0x80` corresponds to the Ping Echo Request.

Question 5.3.2 page 73 Define a compression rule allowing any host on the Internet to ping the device.

I

Question 5.3.3 page 75 Modify the previous rule to allow only the App to ping the device, and limit the Sequence Number to 4 bits.

In this rule, the App IPv6 prefix and IID are checked with the equal/not-sent MO/CDA and the `ICMPv6.SEQNO` uses MSB(12)/LSB to send a residue of 4 bits.

Listing 10.10 – icmp-single.json

```
{
   "DeviceID" : "udp:10.0.0.20:8888",
   "SoR" : [
         {
            "RuleID": 6,
            "RuleIDLength": 3,
            "Compression": [
                {"FID": "IPV6.VER", "TV": 6, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.TC",  "TV": 0, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.FL",  "TV": 0, "MO": "ignore","CDA": "not-sent"},
                {"FID": "IPV6.LEN",          "MO": "ignore","CDA": "compute-length"},
                {"FID": "IPV6.NXT", "TV": 58, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.HOP_LMT", "TV" : 255,"MO": "ignore","CDA": "not-sent"},
                {"FID": "IPV6.DEV_PREFIX","TV": "AAAA::/64",
                                           "MO": "equal","CDA": "not-sent"},
                {"FID": "IPV6.DEV_IID", "TV": "::1","MO": "equal","CDA": "not-sent"},
                {"FID": "IPV6.APP_PREFIX","TV": "2001:0:0:1::/64",
                                           "MO": "equal","CDA": "not-sent"},
                {"FID": "IPV6.APP_IID", "TV": "::15","MO": "equal","CDA": "not-sent"},
                {"FID": "ICMPV6.TYPE",  "TV": 128, "MO": "equal","CDA": "not-sent"},
                {"FID": "ICMPV6.CODE",  "TV": 0,   "MO": "equal","CDA": "not-sent"},
                {"FID": "ICMPV6.CKSUM", "TV": 0,   "MO": "ignore","CDA": "compute-checksum"},
                {"FID": "ICMPV6.IDENT", "TV": 0,   "MO": "ignore","CDA": "value-sent"},
                {"FID": "ICMPV6.SEQNO", "TV": 0,   "MO": "MSB", "MO.VAL" : 12,
                                            "CDA": "LSB"},
                {"FID": "ICMPV6.PAYLOAD", "TV": 0,"MO": "ignore","CDA": "value-sent"}
            ]
         }
   ]
 }
```

Question 5.3.4 page 75 What will be the Sequence Number residue size to avoid padding at the end of the SCHC message.

We saw that the padding was 5 bit long, so if we fix the MSB comparison to 11 bits, the 5 remaining bits will force the alignment of the SCHC residue.

Question 6.2.1 page 85 If the Device includes a new sensor, for example, CO2 level, what is the impact on the description given in the table 6.1 on page 87 ?

A new URI is added to the list ; this will not have an impact on compression since 2 bits are needed to code this residue and only 3 values were previously defined.

Question 6.2.2 page 92 Introduce some errors in the `render_post` method, such as a division by 0. What will be the notification code ?

The following capture shows the error message.

```
0x0000:  6009 6329 000c 1140 2001 0000 0000 0001  `.c)...@........
0x0010:  0000 0000 0000 0015 aaaa 0000 0000 0000  ................
0x0020:  0000 0000 0000 0001 1633 150c 000c 6d8f  .........3....m.
0x0030:  50a0 4ba5                                P.K.
```

The notification `0xA0` can be written `0b101_00000` or 5.00.

Question 6.2.3 page 92 Suppose that an old CoAP server does not recognize the *No Response* option. What are the consequences for the Device ? How do we solve this ?

If the *No Response* option is ignored, since in rule `255/8` in Listing 6.1 on page 86, the CDA for `CoAP.CODE` is `ignore`, every `2.04` notification will be compressed to rule `255/8`. The device will believe that every request leads to an error. One solution to protect against this behavior is to use a matching list as an MO, listing all 4.0x and 5.0x on the TV and still setting `not-sent` to the CDA.

Question 6.2.4 page 93 In this scenario, the notification is expected to arrive 1 second after the transmission of the request. In an LPWAN environment, the downlink message may be delayed to respect a Duty Circle. Does the system continue to work ? How do we solve this problem ?

The system will not work since there is no way to link the notification with the request. The usage of CON messages instead of NON, forces an ACK for each request. The only way to establish the link between a request and a response. The only way to make a link between a request and a notification, when NON messages are used, is to take a Token. Since the message ID cannot be removed, this adds one byte to the residue [3]

Question 6.3.1 page 98 The `underwater.json` Set of Rules (cf. page 94) contains a single rule used bidirectionally to compress request and response. Modify this Set of Rules, to handle specifically the notification errors.

What is the impact on traffic ?

We modify rule 1/3 and fix the value for `CoAP.CODE` to `2.05` for the GET response. We also add a new rule to handle other notifications than `2.05`. To avoid padding the rule length can be 8 bit long, the new rule contains : ruleID (8 bits), App Port (16 bits), Code (8 bits), MID (16 bits) and Token (16 bits).

This new rule allows 1 byte reduction of Rule 1/3.

Question 7.2.1 page 105 What will be the maximum FCN value if this field is coded 4 bits ? What will be the All-1 value of the FCN ?

Frag gets numbers 14 to 0 and the All-1 FCN is 15.

Question 7.2.2 page 105 What append if the All-1 fragment is lost ?

This SCHC message and the following one will be discarded, as the second one will be joined and the RCS of the second one will not correspond to the concatenation.

Question 7.6.1 page 111 Sigfox network uses an L2 frame with a maximum of 12 bytes. The FCN field size is 4 bits long. The Rule ID is 2 bits long and there is no DTAG. Verify that a window field size of 3 bits is sufficient to transmit a compressed frame of 1280 bytes.

3. Taking less than a byte, will lead to padding.

The calculation is not straightforward, as it requires knowledge of the SCHC header size to determine the number of bits per fragment, and understanding the total number of fragments is necessary to determine the window field size and thus the size of the SCHC fragmentation header.

Assuming initially, to estimate the quantity of fragment, that the window size is 3 bits. Given that the Fragmentation header includes the Rule ID, the Window, and the FCN field, its total size amounts to $2+3+4=9$ bits.

It remains $12\times 8-9=87$ bits to carry a fragment. We need $\lceil\frac{1280\times 8}{87}\rceil = 118$ fragments (117 of 87 bits and the last one of 61).

Each window contains $2^4-1=15$ fragments, so we need $\lceil\frac{118}{15}\rceil = 8$ windows. This 3 bits in the window field, we can number them.

Question 7.6.2 page 111 With the same hypothesis as in the previous question, what is the window field size if the FCN field size is set to 2 ? and to 5 ?

The FCN field size is set to 2, each window contains 3 fragments. We take the assumption that the window field size is on 4 bits. The fragment header size is the same as in the previous question $2+4+2=8$, so each fragment contains $12\times 8-8=88$ bits. The number of fragments is $\lceil\frac{1280\times 8}{88}\rceil = 117$ fragments, corresponding to $\lceil\frac{117}{3}\rceil = 39$. With 4 bits for the Window field size, it is not enough.

The windows field must be at least on 6 bits. So, a fragment contains 86 bits, leading to 120 fragments and 40 windows.

The FCN field size is set to 5, each window contains 31 fragments. Let's keep the window field size on 3 bits. The fragmentation header size is $2+3+5=10$ bits and 86 bits of payload. We have 120 fragments to send, and 4 windows. The window field size could be reduced to 2 bits.

Question 8.2.1 page 120 When examining the Rule Manager rules display, how large is the FCN field for rule 4/3 ?

openSCHC set by default the FCN field by default to 1 bits. It also includes a 2 bit long Dtag field.

Question 8.2.2 page 121 Why the Application do not receive a response to its ping message ?

There is no fragmentation rule defined for uplink traffic.

Question 8.2.3 page 121 Edit the rules on the Device and in the Core to fix the problem.

A Fragmentation rule has been defined for downlink traffic. The device also needs an uplink fragmentation rule to send back its answer.

SCHOOL

11. OpenSCHC Identifiers

11.1 Field Id

RFC 8724	**openSCHC**	**RFC 9363** (YANG DM)	SID	**Length (bits)**	**Position**	RFC
IPv6 Version	IPV6.VER	fid-ipv6-version		4	1	RFC 8200
IPv6 Diffserv	IPV6.TC	fid-ipv6-trafficclass		8	1	
IPv6 Flow Label	IPV6.FL	fid-ipv6-flowlabel		20	1	
IPv6 Length	IPV6.LEN	fid-ipv6-payload-length		16	1	
IPv6 Next Header	IPV6.NXT	fid-ipv6-nextheader		8	1	
IPv6 Hop Limit	IPV6.HOP_LMT	fid-ipv6-hoplimit		8	1	
IPv6 DevPrefix	IPV6.DEV_PREFIX	fid-ipv6-devprefix		64	1	
IPv6 DevIID	IPV6.DEV_IID	fid-ipv6-deviid		64	1	
IPv6 AppPrefix	IPV6.APP_PREFIX	fid-ipv6-appprefix		64	1	
IPv6 AppIID	IPV6.APP_IID	fid-ipv6-appiid		64	1	
UDP DevPort	UDP.DEV_PORT	fid-udp-dev-port		8	1	
UDP AppPort	UDP.APP_PORT	fid-udp-app-port		8	1	
UDP Length	UDP.LEN	fid-udp-length		8	1	
UDP checksum	UDP.CKSUM	fid-udp-checksum		8	1	

TABLE 11.1 – Field ID definition in different implementations

RFC 8724	openSCHC	RFC 9363 (YANG DM)	SID	Length (bits)	Position	RFC
CoAP Version	COAP.VER	fid-coap-version		2	1	
CoAP Type	COAP.TYPE	fid-coap-type		2	1	
CoAP Token Length	COAP.TKL	fid-coap-tkl		4	1	
CoAP Code	COAP.CODE	fid-coap-code		8	1	
CoAP Message ID	COAP.MID	fid-coap-mid		16	1	
CoAP Token	COAP.TOKEN	fid-coap-token		tlk	1	
CoAP if-match	COAP.If-Match	fid-coap-option-if-match		var	*	
CoAP Uri-host	COAP.Uri-Host	fid-coap-option-uri-host		var	1	
CoAP Etag	COAP.ETag	fid-coap-option-etag		var	*	
CoAP If-None-Match	COAP.If-None-Match	fid-coap-option-if-none-match		0	1	
CoAP Observe	COAP.Observe	fid-coap-option-observe		var	1	
CoAP Uri-Port	COAP.Uri-Port	fid-coap-option-uri-port		var	1	
CoAP Location-Path	COAP.Location-Path	fid-coap-option-location-path		var	*	
CoAP Uri-Path	COAP.Uri-Path	fid-coap-option-uri-path		var	*	
CoAP Content-Format	COAP.Content-Format	fid-coap-option-content-format		var	1	
CoAP Max-Age	COAP.Max-Age	fid-coap-option-max-age		var	1	
CoAP.Uri-Query	COAP.Uri-Query	fid-coap-option-uri-query		var	*	
CoAP Max-Age	COAP.Max-Age	fid-coap-option-max-age		var	1	
CoAP Accept	COAP.Accept	fid-coap-option-accept		var	1	
CoAP Location-Query	COAP.Location-Query	fid-coap-option-location-query		var	1	
CoAP Block2	COAP.Block2	fid-coap-option-block2		var	1	
CoAP Block1	COAP.Block1	fid-coap-option-block1		var	1	
CoAP Size2	COAP Size2	fid-coap-option-size2		var	1	
CoAP Proxy-Uri	COAP.Proxy-Uri	fid-coap-option-proxy-uri		var	1	
CoAP Proxy-Scheme	COAP.Proxy-Scheme	fid-coap-option-proxy-scheme		var	1	
CoAP Size1	COAP.Size1	fid-coap-option-size1		var	1	
CoAP.No-Response	COAP.No-Response	fid-coap-option-no-response		8	1	
CoAP OSCORE_flags		fid-coap-option-oscore-flags		var	1	
CoAP OSCORE_piv		fid-coap-option-oscore-piv		var	1	
CoAP OSCORE_kid		fid-coap-option-oscore-kid		var	1	
CoAP OSCORE_kidctx		fid-coap-option-oscore-kidctx		var	1	

TABLE 11.2 – Field ID definition in different implementations

Index

Symbols

Uri-path 136
ACK 20
AsyncSniffer 117
CON 20
ICMPV6.CHECKSUM 74
ICMPV6.SEQNO 74
ICMPv6.IDENT 74
IPV6.APP_IID 56
IPV6.FL 56, 73
IPV6.HOP_LMT 73
IPV6.LENGTH 74
IPV6.NXT 73
IPV6.TC 56
IPV6.VERSION 73
NON 20
NoCompression 38
RST 20
RuleIDLength 49
RuleIDValue 49
Uri-Host 27
Uri-Path 27
Uri-Query 27
aoicoap 31
pprint 52
2.04 136
2.05 136

A

AA 100
Accept 27, 28
addresses 74
aiocoap 84, 90
AoE 100
appIID 48

B

bi-directional 45

C

CBOR 29
CDA 47–49
Compression 38, 49
compute-* 61
compute-checksum 48
compute-length 48

CON . . . 31
Content-Format . . . 27, 28
CORE . . . 68

D

DeviceID . . . 40, 76
devIID . . . 48
DI . . . 49
DiffServ . . . 15
Direction Indicator . . . 45, 64, 79
Directional Indicator . . . 94
down-link . . . 45
duty cycle . . . 84

E

ECN . . . 15
Equal . . . 47
ETag . . . 27

F

failed_field . . . 55
FCNSize . . . 104
FID . . . 49
Field ID . . . 64
Field Length . . . 45
Field Position . . . 45, 64
FL . . . 49
Flow Label . . . 15
Fragmentation . . . 38
FV . . . 56

G

gen_parameters . . . 52

H

hexdump . . . 53
Hop Limit . . . 15

I

ICMP . . . 13
ICMPv6 . . . 15
ICMPV6.PAYLOAD . . . 72
ICMPV6.TYPE . . . 79, 81
If-Match . . . 27
If-None-Match . . . 27
iface . . . 71
Ignore . . . 46
ignore . . . 73
IID . . . 74, 132
IPv4 . . . 13
IPv6 . . . 13
IPV6.FL . . . 133

J

JSON . . . 29

L

L2 Word . . . 102
link-local . . . 15
Location-Path . . . 27
Location-Query . . . 27
LoRaWAN . . . 41
LSB . . . 48
LwM2M . . . 29

M

mapping-sent . . . 48

Match-Mapping . . . 47
Max-Age . . . 27
MO . . . 46, 49
MO.VAL . . . 49
Module fPython
  gen_parameters.py . . . 49
Module Python
  aiocoap
    add_resource, 91
    render_, 90
    Resource, 90
  compr_core.py
    Compressor, 58
  compr_parser
    Parser, 52, 71
  compress
    compress, 58
  Decompressor
    compr_core, 61
  gen_bitarray
    BitBuffer, 97
  gen_bitarray.py
    BitArray, 59
  gen_rulemanager
    Add, 39, 41
    Print, 39
    RuleManager, 39, 54
  gen_rulemanager.py
    FindRuleFromSCHCpacket, 81
  get_reamaining_content
    BitBuffer, 62
  get$_bits$
    r, 97
  protocol.py
    schc_recv, 82
    schc_send, 76, 77
    SCHCProtocol, 77
  RuleManager
    FindRuleFromPacket, 55
  scapy
    show, 53
    sniff, 71, 82
MOOC-2 . . . 70
MSB . . . 47, 48, 131

N

NDP . . . 70
Next Header . . . 15
No-Ack . . . 40
No-Response . . . 27, 64, 85
NoAck . . . 100
NoCompression . . . 38
NON . . . 85
not-sent . . . 48, 73

O

Observe . . . 23, 27, 30
OMA . . . 29
openSCHC . . . 47, 65

P

padding . . . 102
Parser . . . 52
Payload Length . . . 15
Payload Marker . . . 64
Payload marker . . . 28
pcap . . . 15, 51
POST . . . 84
prn . . . 71
Programmes
  coap-server.py . . . 90
  core.py . . . 89, 95
  dev-lpwan.py . . . 88, 92
  pcap_match.py . . . 54
  pcap_parse.py . . . 51, 54
  pcap_read.py . . . 17
  ping_core.py . . . 82
  ping_packet_descr.py . . . 70, 73
  ping_packet_rule.py . . . 72, 77
Programmes micro-python
  coap-server.py . . . 90, 91

- ping$_{d}ev.py$ . . . 80
- ping$_{d}ev1.py$ . . . 117, 118

Programmes Python
- coap-client.py . . . 96
- dev-server.py . . . 97
- pcap$_{c}ompress.py$ . . . 58
- pcap$_{d}ecompress.py$ . . . 61
- pcap$_{f}lowlabel.py$ . . . 126
- pcap$_{m}atch.py$ . . . 54, 55
- pcap$_{p}arse.py$ . . . 52
- pcap$_{r}ead.py$ . . . 17
- pcap$_{r}ead_{c}oap.py$ . . . 127
- ping$_{c}ore.py$ . . . 77
- ping$_{c}ore1.py$ . . . 82
- ping$_{p}acket_{d}escr.py$ . . . 71
- ping$_{p}acket_{r}ule.py$ . . . 73
- rm1.py . . . 39
- rm2.py . . . 40

Proxy-Scheme . . . 27
Proxy-Uri . . . 27

R

RCS . . . 100
rdpcap . . . 52
render_post . . . 91
ReSeT . . . 31
residue . . . 74
RFC 2474 . . . 15
RFC 3168 . . . 15
RFC 6437 . . . 63
RFC 7252 . . . 20
RFC 7641 . . . 30
RFC 7967 . . . 29
RFC 8132 . . . 21
RFC 8200 . . . 14, 99, 145
RFC 8724 . 35, 44, 46, 47, 49, 64, 100, 102, 104, 106, 109, 145, 146
RFC 8824 . . . 67
RFC 9011 . . . 112
RFC 9363 . . . 41, 44, 48, 67, 145, 146
RFC 9441 . . . 111
rule ID . . . 35
Rule Manager . . . 38, 54, 73
RuleID . . . 37
RuleIDLength . . . 37

Rules
- icmp-bi.json . . . 79
- icmp-single.json . . . 139
- icmp.json . . . 73
- ipv6-bi . . . 134
- ipv6-sol . . . 57
- ipv6-sol-bi-fl . . . 135
- ipv6-sol-bi.json . . . 133
- IPv6.json . . . 51
- lpwan.json . . . 86
- residue1 . . . 138
- residue2 . . . 138
- rule$_{c}oap1$ . . . 137
- trace-bitbuffer . . . 134
- underwater.json . . . 94

S

Scapy . . . 16
scapy . . . 52
SCHC instance . . . 34
SenML . . . 29
Set of Rules . . . 34, 40
show_diff . . . 61
Sigfox . . . 102
Size1 . . . 27

T

Target Value . . . 47
tcpdump . . . 70
Tile . . . 112
tkl . . . 65
TLV . . . 29
Token . . . 23, 31
token . . . 65
Token Length . . . 24, 65

Traffic Class 15
TV 46, 49

U

UDP tunnel 40
up-link 45
Uri-Host 27
Uri-host 136
Uri-Path 27
Uri-path 64
Uri-Port 27
Uri-Query 27
Uri-query 64

V

value-sent 48
var 65
Version 15

W

WINDOW_SIZE 104