THE BOOK OF SCHC

Paul OULTRY

IMT ATLANTIQUE

Published 18 août 2022

Table of Contents

Acronyms

AS Application Server
CBOR Concise Binaire Object Representation
CoAP Constrained Application Protocol
IP Internet Protocol
IPv4 Internet Protocol version 4
IPv6 Internet Protocol version 6
LPWAN Low Power Wide Area Network
LNS LoRaWAN Network Server
RTT Round Trip Time

1. Getting started with openSCHC

OpenSCHC provides for an easy way to understand and experiment with both the compression/-decompression mechanism and the fragmentation protocol of SCHC. We start by illustrating the compression and fragmentation processes as defined in RFC 8724. To keep things simple, openSCHC does not have to run on an actual LPWAN network, experiments can be run on a regular network using UDP tunnels.

1.1 Installation

1.1.1 Downloading the code

In order to download openSCHC, just clone the GitHub openschc repository onto your computer,

```
> git clone git@github.com:openschc/openschc.git
```

and select the "scapy" branch.

```
> git checkout scapy
```

1.1.2 Getting IPv6 connectivity

We want to try out the openSCHC compression and fragmentation features on IPv6 traffic (as illustrated in RFC 8724). We therefore need to make our machine able to receive IPv6 traffic. Furthermore, we would like our machine to be able to receive IPv6 traffic destined to several IPv6 hosts (see 1.2).

A simple way to get a global IPv6 prefix is to use the services of a tunnel broker such as Hurricane Electric, see Figure 1.1. Hurricane Electric

To set up the IPv6 tunnel :

— go to the website `https://tunnelbroker.net/`,
— create an account if needed or just log in,
— select *Create another Tunnel*,

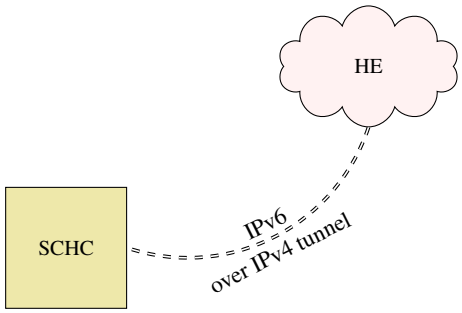

FIGURE 1.1 – setting up IPv6 connectivity

— enter the IPv4 address of the machine running openSCHC and a location,

Once the tunnel is created, you will find some configuration examples, which will help you configuring your machine.

netplan For Ubuntu 20, it will be in the `/etc/netplan` directory :

```
> cat /etc/netplan/50-cloud-init.yaml
# This file is generated from information provided by
# the datasource.  Changes to it will not persist across an instance.
# To disable cloud-init's network configuration capabilities, write a file
# /etc/cloud/cloud.cfg.d/99-disable-network-config.cfg with the following:
# network: {config: disabled}
network:
    ethernets:
        ens3:
            dhcp4: true
            match:
                macaddress: fa:16:3e:e9:db:5d
            addresses:
            - "2001:41d0:404:200:0:0:0:3a86/64"
            gateway6: "2001:41d0:0404:0200:0000:0000:0000:0001"
            set-name: ens3
    version: 2
    tunnels:
      he-ipv6:
        mode: sit
        remote: 216.66.87.102
        local: 51.91.121.182
        addresses:
          - "2001:470:1f20:1d2::2/64"
        gateway6: "2001:470:1f20:1d2::1"
```

Type the following command to validate your new network configuration :

```
>sudo netplan try
Warning: Stopping systemd-networkd.service, but it can still be activated by:
  systemd-networkd.socket
Do you want to keep these settings?


Press ENTER before the timeout to accept the new configuration


Changes will revert in 111 seconds
Configuration accepted.
>sudo netplan apply
```

and your machine interfaces should now look this :

```
> ifconfig
ens3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 51.91.121.182  netmask 255.255.255.255  broadcast 0.0.0.0
        inet6 2001:41d0:404:200::3a86  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::f816:3eff:fee9:db5d  prefixlen 64  scopeid 0x20<link>
        ether fa:16:3e:e9:db:5d  txqueuelen 1000  (Ethernet)
        RX packets 9543878  bytes 1758163442 (1.7 GB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 9253880  bytes 1479370777 (1.4 GB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

he-ipv6: flags=209<UP,POINTOPOINT,RUNNING,NOARP>  mtu 1480
        inet6 fe80::335b:79b6  prefixlen 64  scopeid 0x20<link>
```

```
        inet6 2001:470:1f20:1d2::2  prefixlen 64  scopeid 0x0<global>
        sit  txqueuelen 1000  (IPv6-in-IPv4)
        RX packets 782095  bytes 447475094 (447.4 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 71534  bytes 6022694 (6.0 MB)
        TX errors 5  dropped 0 overruns 0  carrier 5  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 7212  bytes 657864 (657.8 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 7212  bytes 657864 (657.8 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Note that, in this example, the machine is multi-homed : the `ens3` interface is configured with the regular IPv6 address and the `he-ipv6` interface is configured with the Hurricane Electric IPv6 address.

1.2 Building our Test Network

We want to reproduce a typical LPWAN network topology : an Internet-based application server communicates with IoT devices via an LPWAN Network Server (see RFC 8376). We have already configured our SCHC instance on the "Core" side. On a second machine that will represent the IoT device, let's now install openSCHC as well.

The topology described in Figure 1.2 will be used throughout the rest of the tutorial. The compressed and fragmented SCHC packets will be tunneled over UDP across the Internet between the Device SCHC instance and the Core SCHC instance, as if they were carried over an LPWAN network.

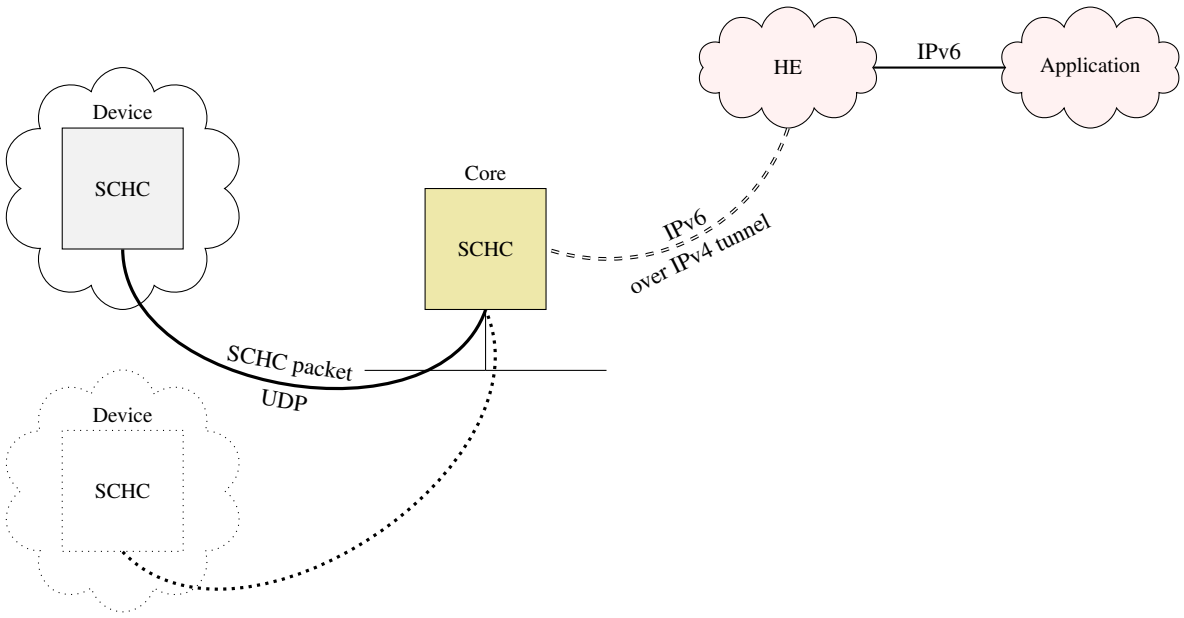

FIGURE 1.2 – Test Network Topology

The "Core" SCHC instance will act as a router between the IPv6 network and the constrained network, illustrated in our example by the UDP tunnel (but it could be made to be a real LPWAN network).

SCHC Packet

As per RFC 8724, the messages exchanged between SCHC entities are called "SCHC Packets".

uplink

downlink

Likewise, the LPWAN traffic from the device to the core is called "uplink", and the reverse traffic is called "downlink" .

application

One or several applications will communicate with the devices.

2. Installing SCHC rules in openSCHC

SCHC is a protocol taking place between two SCHC entities connected via a link. Usually, one SCHC entity is implemented in a constrained end-device and the other one is implemented in a core equipment acting as a router. To allow the pair to perform the same action, or more precisely an action on one end and the inverse action and the other end, e.g., compression and decompression, or fragmentation and reassembly, a common set of rules must be shared between these two entities.

The distinction between a device and a core tells which communication is uplink (from the device to the core) and which is downlink (opposite direction). The device usually contains a small set of compression of fragmentation rules tailored to the traffic generated and received by that device only. Conversely, the core instance of SCHC must know all the devices' rule sets. Even if all the devices happen to be identical, the SCHC core treats each device as if it were unique.

2.1 Rule ID's

Rules are identified by a rule ID. A rule ID is a binary sequence that must be unique within the scope of the link between a device SCHC instance and a core SCHC instance. RFC 8724 does not specify the length of the rule ID's : the lengths are chosen as the rules are defined, 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 instance :

rule ID

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

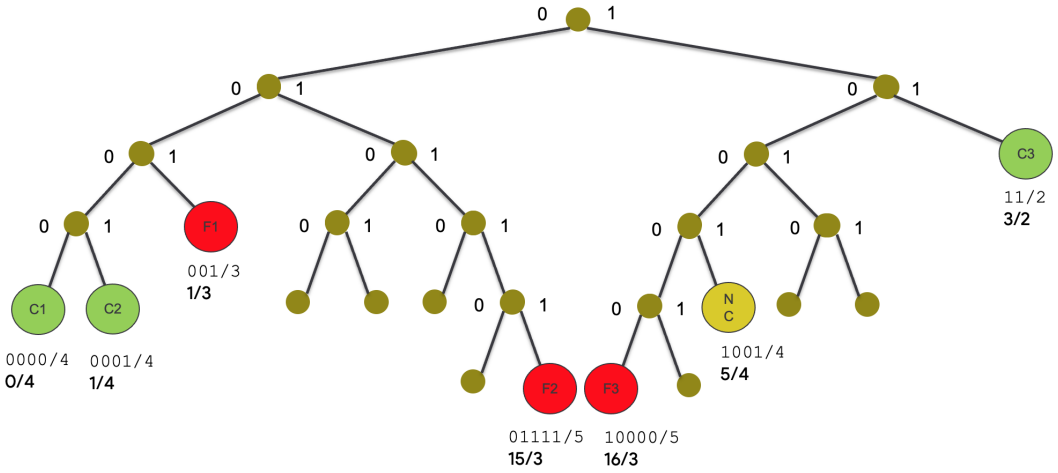

FIGURE 2.1 – Example of binary tree associated to rule IDs

Pay attention that this notation can be misleading : for example, `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`).

2.2 Rules structure

In the openSCHC implementation of SCHC, a rule is described with a JSON object, and the rule ID is expressed with the decimal notation introduced above :

```
{
"RuleID" : 12,
"RuleIDLength" : 4
}
```

Compression

There are three different rule formats. The first one is used for Compression Rules, which contain the `Compression` keyword followed by an array, as shown in the minimal example below :

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

Fragmentation

The second format is used for Fragmentation rules, which contain the `Fragmentation` keyword followed by an object describing the fragmentation parameters :

```
{
"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.

An observant reader may have noticed that the fragmentation rule description includes a direction indicator. Indeed, SCHC compression rules are bi-directional (the same rule can be used to compress a packet flowing from a device to the core or from the core to the device), but fragmentation rules are oriented (they only apply to either uplink of downlink traffic).

2.3 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 rules 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 more compact format than JSON,
- find in the store a valid rule to compress or fragment an incoming packet with, [1]
- retrieve a rule from the store by its rule ID.

2.3.1 Adding Rules into the Rule Manager

Listing 2.1 shows a simple python program used to manage the two rules described in 2.2.

Listing 2.1 – rm1.py

```
1  from gen_rulemanager import *
2
3  RM = RuleManager()
4
5  rule1100 =   {
6    "RuleID" : 12,
7    "RuleIDLength" : 4,
8    "Compression" : []
9  }
10
11 rule001100 =   {
12   "RuleID" : 12,
13   "RuleIDLength" : 6,
14   "Fragmentation" :  {
```

1. TODO : find a good rule, or better yet, the best rule, instead of the first valid one in the store.

```
    "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 into 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
no Ack used here, the Rule Manager added some default parameters corresponding to the no Ack behavior of SCHC.

2.3.2 Adding a Set of Rules

A device should contain a set of rules related to compression and fragmentation. In openSCHC,
Set of Rules the SoR (Set of Rules) is a JSON array. The following program has the same behavior as the one shown in 2.3.1, but the rules are added with a single method call, using an array.

Listing 2.2 – 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()
```

2.3.3 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. Conversely, 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 1.1 on page 6), 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. A full DeviceID would therefore be, for instance, `udp:83.199.24.39:8888` [2]. UDP tunnel
- for LoRaWAN (*not yet implemented*), the technology keyword is `lorawan` and the identifier is the *devEUI*. LoRaWAN

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

```
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" : [ ..... ]
}
```

2.3.4 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 instance :

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

2.4 Conclusion

We have reviewed how rules are structured and how they can be added to the rule manager store. Let's now try to compress and fragment some traffic.

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

3. Compressing ping6 messages

3.1 Introduction

ICMPv6 Echo Request

In this chapter, we are going to setup the compression of an ICMPv6 Echo Request message. Such a message is easy to generate from the command line of any IPv6-enabled computer, using the `ping6` command. It is composed of an IPv6 packet header, followed by the ICMPv6 Echo Request header and some data bytes.

```
Internet Protocol Version 6, Src: 2a01:cb08:903a:bd00:a8c4:5c6d:c2b5:84be, Dst:
2001:470:1f21:1d2::1
    0110 .... = Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
        .... 0000 00.. .... .... .... .... .... = Differentiated Services Codepoint: Default (0)
        .... .... ..00 .... .... .... .... .... = Explicit Congestion Notification: Not ECN-Capable Transport (0)
    .... .... .... 1000 0000 1000 0000 0000 = Flow Label: 0x80800
    Payload Length: 16
    Next Header: ICMPv6 (58)
    Hop Limit: 56
    Source: 2a01:cb08:903a:bd00:a8c4:5c6d:c2b5:84be
    Destination: 2001:470:1f21:1d2::1

Internet Control Message Protocol v6
    Type: Echo (ping) request (128)
    Code: 0
    Checksum: 0x3982 [correct]
    [Checksum Status: Good]
    Identifier: 0x70ef
    Sequence: 699

    Data (8 bytes)
        Data: 609ab0db000aecb7
        [Length: 8]

0000  60 08 08 00 00 10 3a 38 2a 01 cb 08 90 3a bd 00   `.....:8*....:..
0010  a8 c4 5c 6d c2 b5 84 be 20 01 04 70 1f 21 01 d2   ..\m.... ..p.!..
0020  00 00 00 00 00 00 00 01 80 00 39 82 70 ef 02 bb   ..........9.p...
0030  60 9a b0 db 00 0a ec b7
```

The following compression rule is able to compress that traffic.

Listing 3.1 – ping-comp-rule

```
{
    "DeviceID" : "udp:83.199.24.39:8888",
    "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" : 64,"MO": "ignore","CDA": "not-sent"},
            {"FID": "IPV6.DEV_PREFIX","TV": "2001:470:1F21:1D2::/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",  "DI": "DW", "TV": 128,"MO": "equal","CDA": "not-sent"},
            {"FID": "ICMPV6.TYPE",  "DI": "UP", "TV": 129,"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"}

          ]
    } ] }
```

3.2 More details on compression rules

A rule contains a list of fields descriptors. Each field descriptor includes :

- a Field ID : openSCHC uses a string structured with the protocol name (IPV6 and ICMPV6 in the rule above) and the field name for that protocol. The key for the Field ID is FID. — Field ID, FID
- an optional Target Value (TV) : it contains the value(s) that the rule expects to find in the field of the incoming packet. — Target Value
- a Direction Indicator (DI) : this specifies the traffic direction (uplink, downlink or both) that this descriptor applies to. In the above example, the ICMPV6.TYPE field descriptor appears twice in the rule, the first one is intended to match downlink traffic and the second one matches uplink traffic. When the direction is not specified, openSCHC considers that the field descriptor applies to both directions. — Direction Indicator, ICMPV6.TYPE
- a Matching Operator (MO) : this specifies how to compare the Target Value (TV) in the field descriptor and the value of the field (FV) of the packet to be compressed. RFC 8724 defines 4 MO : — Matching Operator
  - ignore : the comparison always returns "true". — ignore
  - equal : the result of the comparison is true if the value contained in the TV is equal to the value contained if the FV. OpenSCHC allows for comparison between integers and between strings. — equal
  - MSB : the comparison is true if the first bits of the TV are equal to the first bits of the FV. The number of bits to be compared is specified in the rule using the MO.val keyword. For string comparisons, openSCHC mandates that the length be a multiple of 8 bits. — MSB
  - match-mapping : the TV contains an array of values. The comparison is true if an element of TV is equal to FV. — match-mapping

  A rule is valid to be used for compression if there is a one-to-one correspondence between the fields in the incoming packet and the field descriptors in the rule, and if all the field descriptor comparisons return true.
- a Compression Decompression Actions (CDA) : when a rule is selected for compression, the CDA is applied to the packet field. RFC 8724 defines several CDAs, among them : — Compression Decompression Actions
  - not-sent : the FV is not sent, the decompressor will use the TV stored in the rule to recover the field value. — not-sent
  - value-sent : the value is explicitely sent as a compression residue. The decompressor will use the residue to rebuid the field value. — value-sent

LSB

— `LSB` : the least significant bits are sent as a compression residue. The decompressor will concatenate the `TV` and the residue to rebuild the field. LSB can only be used with an MSB MO. No argument is needed for LSB. The LSB size is defined from the original field size and the MSB size.

mapping-sent

— `mapping-sent` : this CDA can only be used with a match-mapping MO. The index in the TV array is sent as residue. The decompressor will use the index to retrieve the value from the TV array.

compute-*

— `compute-*` : the `FV` is not sent, the decompressor will use a specific algorithm to recover the value. For example, a length field can be recomputed from the undelying layer payload size, of a checksum can be recomputed from the payload if it is ascertained that the lower layer is protected by a stronger integrity mechanism (e.g., a CRC).

3.3 Compressing an ICMPv6 Echo Request packet

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 (the capture above shows a value of `0x80800`) 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 as travelled 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.

addresses

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 addresses fields [1], which map to source and destination according to the traffic direction (uplink or downlink). The rule description can therefore be made independent of the direction.

IID

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

In our scenario, the device address is not sent, since the value is known by both the device and the core SCHC entities. Conversely, we allow any host on the Internet to ping the device [2] : 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 subsume 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, and the Echo Reply value (129) is expected uplink. The ICMPv6 `Code` is expected to be 0 in both directions.

1. the same is true for UDP port numbers.
2. 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.

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 :

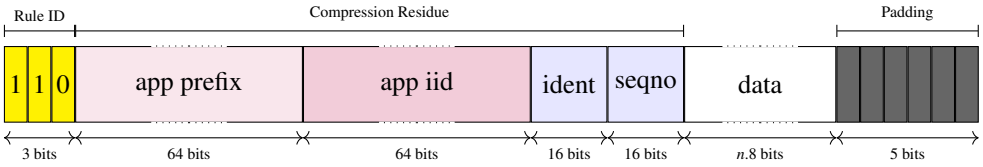

FIGURE 3.1 – ICMPv6 Packet Compressed

The rule ID takes 3 bits, followed by the compression residues, the ICMPv6 payload and some padding bits. 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

3.4 Compression process

In openSCHC, the SCHC Compresssion/Decompression (CD) and Fragmentation/Reassembly (FR) processes are executed by the SCHC Machine.

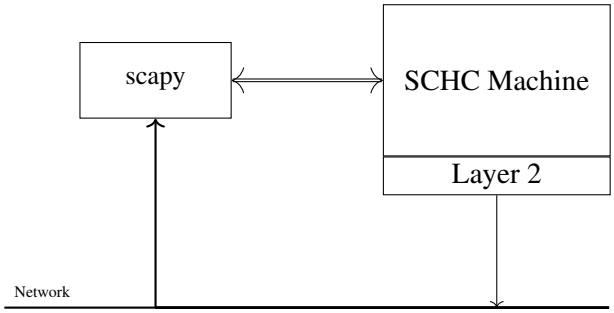

FIGURE 3.2 – Ping gateway architecture

Figure 3.2 shows the architecture of the ping gateway. The Scapy *sniff* function captures the traffic on the network. This traffic comprises IPv6 packets to be compressed and UDP-tunneled SCHC packets to be decompressed. Scapy

3.4.1 The code

The scapy module calls the SCHC Machine, which is in charge of the C/D and F/R processes. Layer 2 allows to inject decompressed IPv6 packets or UDP-tunneled SCHC packets into the internet.

Main program

Listing 3.2 – 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 scapy_connection import *
from gen_utils import dprint, sanitize_value

import pprint
import binascii
import socket
import ipaddress
```

This program `ping_core1.py` shows how to compress ICMPv6 packets and send them on a UDP tunnel to a device. It starts by importing the scapy modules, some openSCHC modules and some generic python modules. Note the importation of the `scapy_connection` module, located in the working directory. This module implements the scheduler and the methods that allow sending packets.

ping_core1.py

scapy_connection

```
# Create a Rule Manager and upload the rules.
rm = RM.RuleManager()
rm.Add(file="icmp1.json")
rm.Add(file="icmp2.json")
rm.Print()
```

The compression process starts with the creation of a Rule Manager `rm`, followed by importing the rules from the file `icmp.json` (cf. Listing 3.3), which contains the compression rules we discussed above and one NoAck fragmentation rule.

Listing 3.3 – rule icmp1.json

```
{
    "DeviceID" : "udp:83.199.24.39:8888",
    "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": "2001:470:1F21:1D2::/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",  "DI": "DW", "TV": 128,"MO": "equal","CDA": "not-sent"},
                 {"FID": "ICMPV6.TYPE",  "DI": "UP", "TV": 129,"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"}

             ]
          },{
                 "RuleID" : 12,
                 "RuleIDLength" : 11,
                 "Fragmentation" : {
                         "FRMode": "NoAck",
                         "FRDirection": "DW"
                 }
          }
    ]
```

```
}
```

The compression rule must be adapted to your environment : the `DeviceID` must reflect the IPv4 public address of your device and the Target Value in `IPV6.DEV_PREFIX` must be set to the IPv6 prefix of your domain, such as the one allocated by Hurricane Electric.

IPV6.DEV_PREFIX

```
48 # Start SCHC Machine
   POSITION = T_POSITION_CORE
50
   socket_port = 0x5C4C
52 tunnel = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
   tunnel.bind(("0.0.0.0", socket_port))
54
   lower_layer = ScapyLowerLayer(position=POSITION, socket=tunnel, other_end=None)
56 system = ScapySystem()
   scheduler = system.get_scheduler()
```

The position of the SCHC instance needs to be specified. In our example, we define our role as a core. Then, we create the UDP tunnel to be able to receive SCHC packets from devices. The default port number is `0x5C4C`.

A `lower_layer` is created to be used by the SCHC Machine when SCHC packet have to be sent, and a `system` is created to manage events. It includes a scheduler. Both are defined in the `scapy_connection` module.

scheduler

scapy_connection

From that `system` a reference to the `scheduler` is extracted to be used in the `processPkt` method, as a global variable.

```
58 schc_machine = SCHCProtocol(
       system=system,            # define the scheduler
60     layer2=lower_layer,       # how to send messages
       role=POSITION,            # DEVICE or CORE
62     verbose = True)
   schc_machine.set_rulemanager(rm)
```

The previously created classes are regrouped around a SCHC Machine which will process packets for compression and fragmentation. A `schc_machine` instance is created and the previously created rule manager is associated to it.

SCHC Machine

```
sniff(prn=processPkt, iface="ens3") # scappy cannot read multiple interfaces --> use sudo
```

Finally, the scapy `sniff` function is called. This line never returns. The `processPkt` function is called each time a packet is received on interface `ens3` [3].

Frame processing

```
24 def processPkt(pkt):
       """ called when scapy receives a packet, since this function takes only one argument,
26     schc_machine and scheduler must be specified as a global variable.
       """
28
       scheduler.run(session=schc_machine)
```

3. Scapy allows to simultaneously listen to nore than one interface, for example `["ens3", "he-ipv6"]`, but we have experienced that this feature sometime returns errors, therefore we prefer to only listen to interface `"ens3"`, which carries the tunneled IPv6 packets from Hurricane Electrics.

run The `processPkt` function starts by calling the SCHC machine through the `run` method. This function must be called regularly, which is the case when some traffic occurs in the network, so it is important not to filter too much incoming traffic. scapy_scheduler

The functions looks for two types of packets in the incoming traffic :

— UDP-tunneled SCHC packets arriving from devices and destined to the UDP port we defined (`0x5C4C` in our case),

— tunneled IPv6 packets from Hurricane Electric, easily recognisable through the use of IP proto 41.

```
    # look for a tunneled SCHC pkt
32  if pkt.getlayer(Ether) != None: #HE tunnel do not have Ethernet
        e_type = pkt.getlayer(Ether).type
34      if e_type == 0x0800:
            ip_proto = pkt.getlayer(IP).proto
36          if ip_proto == 17:
                udp_dport = pkt.getlayer(UDP).dport
38              if udp_dport == socket_port: # tunnel SCHC msg to be decompressed
                    print ("tunneled␣SCHC␣msg")
40                  schc_pkt, addr = tunnel.recvfrom(2000)
                    other_end = "udp:"+addr[0]+":"+str(addr[1])
42                  print("other␣end␣=", other_end)
                    r = schc_machine.schc_recv(other_end, schc_pkt)
44                  print (r)
```

First, a test is done to ensure that the frame has an Ethernet encapsulation [4]. Then we filter the frames to keep only those with :

— an Ethertype equal to 0x800 indicating an IPv4 packet,

— an IPv4 protocol number equal to 17 indicating an UDP message,

— and a destination port of `0x5C4C` indicating a SCHC packet.

If all these conditions are met, we are sure that the socket has received a packet from a device ; we will see in the following step how to process it.

Let's now focus on the incoming Ping6 request.

```
            elif ip_proto==41:
46              schc_machine.schc_send(bytes(pkt)[34:])
```

If the IP protocol value is 41, then we have a tunneled IPv6 packet over IPv4. The first 34 bytes corresponding to the Ethernet and IPv4 headers are removed, and the resulting IPv6 packet is sent to the SCHC machine for compression with the `schc_send` method. schc_send SCHCProtocol

This function will trigger the emission of one or more SCHC packets :

— if the compressed SCHC packet fits in a single frame of the underlying link technology, it is sent,

— otherwise fragmentation is applied and several SCHC packets are sent.

The function returns :

— `None` if a compression rule and optionally a fragmentation rule were found

4. not really needed since we capture all packets through the ens3 interface. The test would be needed if we were to listen to both the ens3 and he-ipv6 interfaces, since the packets arriving on he-ipv6 do not have an Ethernet encapsulation.

— False if a compression rule or a fragmentation could not be found.

The SCHC packet will be sent using the `scapy_connection` methods. scapy_connection

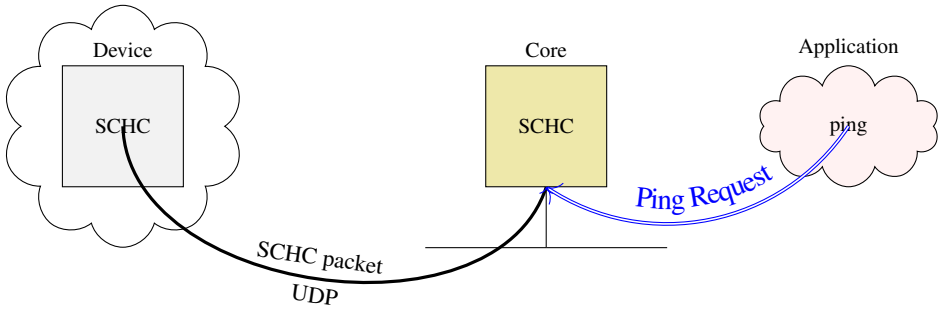

FIGURE 3.3 – SCHC core processing a Ping Request from an Application

3.4.2 Running the code

Start the program, in sudo mode :

```
$ sudo python3.9 ping_core1.py
```

The program will first display the rules after importing them into the store and then the cursor will spin, indicating that the SCHC machine is running [5].

Start pinging the device, from any host on the Internet with IPv6 connectivity, to the IPv6 address of the device defined in the rule. The `-c 1` command line argument limits the number of ping messages.

```
$ ping6 2001:470:1f21:1d2::1 -c 1
```

Of course, there is no answer to the ping at this time. The openSCHC core instance merely displays some messages (they can be turned off by setting the `verbose` argument to False, or by omitting it altogether, when the `schc_machine` object is created).

```
schc recv-from-l3 None None b'‘\x05\x0c\x00\x00\x10:8*\x01\xcb\x08\x90:\xbd\x00I\xe0\xa3\xec\x01Vv\x9c \x01\x04p...'
schc parser {('IPV6.VER', 1): [6, 4], ('IPV6.TC', 1): [0, 8], ('IPV6.FL', 1): [330752, 20]...}
schc compression rule {'RuleID': 6, 'RuleIDLength': 3, 'Compression': [{'FID': 'IPV6.VER', 'FL': 4, ...}
schc compression result b'\xc5\x40\x39\x61\x12\x07\x57\xa0\x09\x3c\x14\x7d\x80\x2a\xce...'/227
schc fragmentation not needed size=227
```

This trace reveals how the openSCHC compression process is divided into several steps :

— parse the packet ; from a sequence of byte received on the network, create a list of fields containing the field identification and their associated value.
— find a valid compression rule ; ask the rule manager to find a rule matching the parsed packet. The rule selection will also provide the device ID.
— apply the compression rule.
— send the SCHC packet to the device ID.

In more details :

5. The rate of received packets determines the spinning speed : the cursor changes appearance every 10 incoming packets.

— The first line (`recv-from-l3`) dumps the original IPv6 packet received by the compressor, corresponding in hexadecimal to :

```
60050c0000103a382a01cb08903abd0049e0a3ec0156769c200104701f2101d2000000000000000001800051fb48b20000609f882600060ed2
```

— The second line (`parser`) shows three elements returned by the parser. The first one is the list of fields, the second one is the data following the list of fields and the third one is a status code. The parsed headers are displayed figure 3.4 :

Listing 3.4 – Parsed IPv6/ICMPv6 header fields

```
{('ICMPV6.CKSUM', 1): [20987, 16],
 ('ICMPV6.CODE', 1): [0, 8],
 ('ICMPV6.IDENT', 1): [18610, 16],
 ('ICMPV6.SEQNO', 1): [0, 16],
 ('ICMPV6.TYPE', 1): [128, 8],
 ('IPV6.APP_IID', 1): [b'I\xe0\xa3\xec\x01Vv\x9c', 64],
 ('IPV6.APP_PREFIX', 1): [b'*\x01\xcb\x08\x90:\xbd\x00', 64],
 ('IPV6.DEV_IID', 1): [b'\x00\x00\x00\x00\x00\x00\x00\x01', 64],
 ('IPV6.DEV_PREFIX', 1): [b' \x01\x04p\x1f!\x01\xd2', 64],
 ('IPV6.FL', 1): [330752, 20],
 ('IPV6.HOP_LMT', 1): [56, 8, 'fixed'],
 ('IPV6.LEN', 1): [16, 16, 'fixed'],
 ('IPV6.NXT', 1): [58, 8, 'fixed'],
 ('IPV6.TC', 1): [0, 8],
 ('IPV6.VER', 1): [6, 4]}
```

As shown in figure 3.4, a header description is a dictionnary where keys are tuple Field ID and position [6], and the value is the tuple containing field value and field size in bits.
The next return element is the data, i.e. the bytes following the parsed header :

```
b'‘\x9f\x88&\x00\x06\x0e\xd2'
```

and the error code is None since the parser recognize all the fields.
This no surprise, the rule 6/3 matches.

— The third line `compression result` gives the SCHC packet. Note the `/227` [7] at the end, indicating the length in bits. Converted in hexadecimal, we have :

```
b'c5403961120757a0093c147d802aced3891640000c13f104c000c1da40'
```

— Finally, no fragmentation is required, so the SCHC packet is directly sent on the tunnel. A frame capture of UDP frame with port 0x5C4C gives :

```
>sudo tcpdump -nXi ens3 udp port 0x5C4C
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens3, link-type EN10MB (Ethernet), capture size 262144 bytes
16:41:48.609729 IP 51.91.121.182.23628 > 83.199.24.39.8888: UDP, length 29
        0x0000:  4500 0039 ac95 4000 4011 751f 335b 79b6  E..9..@.@.u.3[y.
        0x0010:  53c7 1827 5c4c 22b8 0025 1936[c540 3961  S..'\L"..%.6.@9a
        0x0020:  1207 57a0 093c 147d 802a ced3 8e5f c000  ..W..<.}.*..._..
        0x0030:  0c13 fbb5 8000 d951 80]                   .......Q.
```

6. In this example, position is always 1 since no field is repeated several time.

7. 227 % 8 = 3. Since the all the residues are byte alligned, the 3 represent the rule ID length. 5 bits of padding will have to be addeed.

3.5 The Decompression Process

Now that we send SCHC packet to the device, lets process the SCHC packet on this side.

3.5.1 Decompression

Let's do the same operation on the device side. The code is almost the same, as the core SCHC. It is important to note that the rules are excactly the same as the one we used in the core SCHC.

Listing 3.5 – ping_device1.py

```
POSITION = T_POSITION_DEVICE

from requests import get

ip = get('https://api.ipify.org').text

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

device_id = 'udp:'+ip+":"+str(socket_port)
print ("device_id is", device_id)
```

The position is changed to `T_POSITION_DEVICE`. Since the `device_id`, in our example, is based on a public IP address, the call to `https://api.ipify.org` returns the IPv4 public address of the device. We create a UDP socket for port 8888.

```
lower_layer = ScapyLowerLayer(position=POSITION, socket=tunnel, other_end=None)
system = ScapySystem()
scheduler = system.get_scheduler()
schc_machine = SCHCProtocol(
    system=system,            # define the scheduler
    layer2=lower_layer,       # how to send messages
    role=POSITION,            # DEVICE or CORE
    verbose = True)
schc_machine.set_rulemanager(rm)

sniff(prn=processPkt, iface="en0") # scappy cannot read multiple interfaces
```

The `lower_layer`, `system` and `schc_machine` are declared the same way as in the core SCHC. The interface name is adapted to its name on the device side.

```
            if ip_proto == 17:
                udp_dport = pkt.getlayer(UDP).dport
                if udp_dport == socket_port: # tunnel SCHC msg to be decompressed
                    print ("tunneled SCHC msg")
                    schc_pkt, addr = tunnel.recvfrom(2000)
                    r = schc_machine.schc_recv(device_id=device_id, schc_packet=schc_pkt)
                    print (r)
```

It's time now, to look how a SCHC packet is processed. If the IP protocol is 17, the upper layer is UDP. We check the destination port (here 8888) to see if the message is for us. If it is the case, then we can get officially the message from the socket, to clear the buffers. We read the socket and get the message (`schc_pkt`) and the device address (`addr`) which send the SCHC packet.

schc_recv

Then we call the function `schc_recv` with the device ID we computed before and the SCHC packet we just received. This function returns : SCHCProtocol

— `None`. This is the normal behavior when receiving a fragment. Only the last fragment will return the full messages. Intermediary fragments will return this value.
— a byte array with the uncompressed packet.
— `False` when a error occurs.

Since we are just doing compression, the `schc_recv` function returns the full packet. You may notice that some fields have changed. That's the case for IPV6.FL, PV6.HOP_LMT. IPV6.FL PV6.HOP_LMT

3.5.2 Device optimization

In the previous program, we started to reconstruct the packet. It can be then processed normally or even forwarded to another destination. Nevertheless, we can optimize the behavior and process directly the SCHC packet. This simplify the protocol stack implementation in a very constrained device.

Listing 3.6 – extract from rule 6/3 for ping traffic

```
{
    "DeviceID" : "udp:83.199.24.39:8888",
    "SoR" : [
        {
            "RuleID": 6,
            "RuleIDLength": 3,
            "Compression": [
...
                {"FID": "ICMPV6.TYPE",  "DI": "DW", "TV": 128,"MO": "equal","CDA": "not-sent"},
                {"FID": "ICMPV6.TYPE",  "DI": "UP", "TV": 129,"MO": "equal","CDA": "not-sent"},
...
              ]
```

As you have may notice, the ICMPV6.TYPE is defined twice : one for downlink direction, (i.e. from the network to the device) and one in the uplink direction. For the former, the code for an ICMPv6 Echo Request (128) is set and, for the latter it corresponds to the code of an ICMPv6 Echo Reply (129). ICMPV6.TYPE

The SCHC compressed packet do not contain the code itself, and the compression result of the Echo Request and the Echo Reply will look the same. So we can use this property to optimize the code on the device and base the response on the SCHC compressed packet directly, not on the decompressed one. The first step is to identify the rule ID. The rule 6/3 (cf. rule figure 3.3 on page 19) has been associated to the ping traffic. If the rule in the SCHC packet corresponds, we directly send back the SCHC packet to the core, containing in its residue the IPv6 address of the application, the ping identifier and sequence number and the ping payload.

Listing 3.7 – ping_device2.py

```
import gen_rulemanager as RM
from protocol import SCHCProtocol
from scapy_connection import *
from gen_utils import dprint, sanitize_value
from gen_bitarray import *
```

We have to identify the rule in the SCHC packet, we just receive. We need to transform the Byte Array, into a bit array. Therefore, the `gen_bitarray` module is imported [8]. gen_bitarray

8. Other importations will be used when IPv6 packet will be manipulated.

```
                if udp_dport == socket_port: # tunnel SCHC msg to be decompressed
66                  print ("tunneled SCHC msg")
                    schc_pkt, addr = tunnel.recvfrom(2000)
68                  schc_bbuf = BitBuffer(schc_pkt)
                    rule = rm.FindRuleFromSCHCpacket(schc=schc_bbuf, device=device_id)
70                  if rule[T_RULEID] == 6 and rule[T_RULEIDLENGTH]== 3:
                        print ("ping")
72                      tunnel.sendto(schc_pkt, addr)

74                  else:
```

gen_bitarray

The SCHC packet is first transformed into a bit array with the `BitBuffer` constructor.

BitBuffer

gen_rulemanager

We can call the function `FindRuleFromSCHCpacket` with the device ID and the SCHC packet. If a rule ID is found, the function will return it [9].

FindRuleFromSCHCpacket

We check the rule to see if it corresponds to the ping dedicated rule 6/3. We need to test the value and the length, since two different rules may have the same value (remember chapter 2 on page 10).

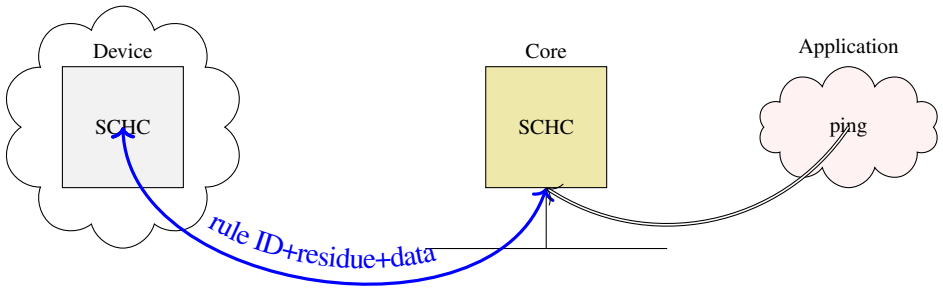

FIGURE 3.4 – SCHC on device processing a Ping Request from an Application and replying

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

3.6 Generating packets

Now, the SCHC core instance receives SCHC packets from the device through the UDP tunnel. We can decompress the SCHC message and send it to the Internet.

Listing 3.8 – ping_core2.py

```
            if ip_proto == 17:
32              udp_dport = pkt.getlayer(UDP).dport
                if udp_dport == socket_port: # tunnel SCHC msg to be decompressed
34                  print ("tunneled SCHC msg")
                    schc_pkt, addr = tunnel.recvfrom(2000)
36                  other_end = "udp:"+addr[0]+":"+str(addr[1])
                    print("other end =", other_end)
```

9. We are very optimistic in this example. If no rule is found, the function will return None and the program will crash.

```
38                      uncomp_pkt = schc_machine.schc_recv(device_id=other_end,
                                                            schc_packet=schc_pkt)
40                      if uncomp_pkt != None:
                            uncomp_pkt[1].show()
42                          send(uncomp_pkt[1], iface="he-ipv6")
                elif ip_proto==41:
44                  schc_machine.schc_send(bytes(pkt)[34:])
```

The processing is almost the same as for the device, if a UDP message with destination port corresponding to the tunnel (i.e. `0x5C4C`) is detected in the packet, then the socket is read. It differs in the way the device is identified. Here, the sender address returned by the socket is used to build the device ID (stored in `other_end` variable).

device ID

The call to the SCHC machine `schc_recv` method may return the uncompressed packet, in the scapy format [10].

schc_recv SCHCProtocol

If the packet is returned, the `show` function displays it and then it is sent on the Hurricane Electric interface.

show scapy

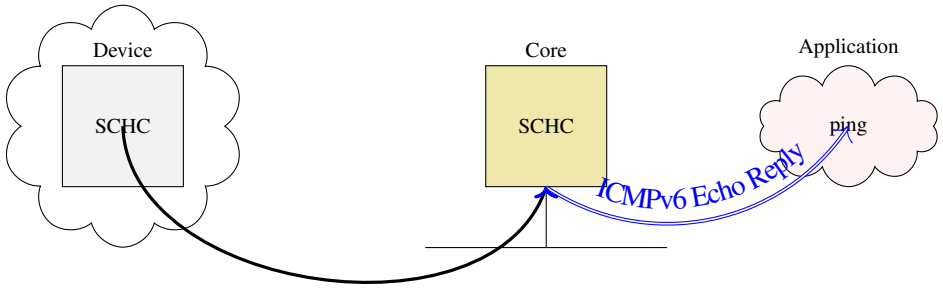

FIGURE 3.5 – SCHC core generating an IPv6 packet

3.7 Execution

3.7.1 On the application

On the application, a regular ping command is typed, which this time returns a message :

```
\$ ping6 -c 1 -s 1 dev2.openschc.net
PING dev2.openschc.net(2001:470:1f21:1d2::2 (2001:470:1f21:1d2::2)) 1 data bytes
9 bytes from 2001:470:1f21:1d2::2 (2001:470:1f21:1d2::2): icmp_seq=1 ttl=244

--- dev2.openschc.net ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
```

Note that the Round Trip Time (RTT) is measured at 0ms since a UDP tunnel is used between the SCHC core and the device. Delay will have been higher if a real Low Power Wide Area Network (LPWAN) network is used.

3.7.2 On the SCHC core

On the SCHC core, OpenSCHC is launched. It loads the rules and wait for incoming IPv6 packets.

10. If the SCHC packet was a fragment, it will have been buffered until the last fragment is received. In that case None would have been returned.

```
\$ sudo python3 ping_core2.py
**************************************
Device: udp:54.37.158.10:8888
/-------------------------\
|Rule 6/3            110  |
|---------------+---+--+--+------------------------------+-------------+----------------\
|IPV6.VER       |  4| 1|BI|                               6|EQUAL        |NOT-SENT        |
|IPV6.TC        |  8| 1|BI|                               0|EQUAL        |NOT-SENT        |
|IPV6.FL        | 20| 1|BI|                               0|IGNORE       |NOT-SENT        |
|IPV6.LEN       | 16| 1|BI|------------------------------|IGNORE       |COMPUTE-LENGTH  |
|IPV6.NXT       |  8| 1|BI|                              58|EQUAL        |NOT-SENT        |
|IPV6.HOP_LMT   |  8| 1|BI|                             255|IGNORE       |NOT-SENT        |
|IPV6.DEV_PREFIX| 64| 1|BI|              200104701f2101d2|EQUAL        |NOT-SENT        |
|IPV6.DEV_IID   | 64| 1|BI|              0000000000000002|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|                             128|EQUAL        |NOT-SENT        |
|ICMPV6.TYPE    |  8| 1|UP|                             129|EQUAL        |NOT-SENT        |
|ICMPV6.CODE    |  8| 1|BI|                               0|EQUAL        |NOT-SENT        |
|ICMPV6.CKSUM   | 16| 1|BI|                               0|IGNORE       |COMPUTE-CHECKSUM|
|ICMPV6.IDENT   | 16| 1|BI|                               0|IGNORE       |VALUE-SENT      |
|ICMPV6.SEQNO   | 16| 1|BI|                               0|IGNORE       |VALUE-SENT      |
\---------------+---+--+--+------------------------------+-------------+----------------/
/-------------------------\
|Rule 12/11     00001100  |
!=========================+=============================================================================\
!v Fragmentation mode : NoAck    header dtag 2 Window  0 FCN  3                      DW v!
!v No Tile size specified                                                              v!
!v RCS Algorithm: RCS_RFC8724                                                          v!
\=========================================================================================/
/-------------------------\
|Rule 13/11     00001101  |
!=========================+=============================================================================\
!^ Fragmentation mode : NoAck    header dtag 2 Window  0 FCN  3                      UP ^!
!^ No Tile size specified                                                              ^!
!^ RCS Algorithm: RCS_RFC8724                                                          ^!
\=========================================================================================/
```

An incoming IPv6 packet arrives, then parsed and compressed, if a rule is found. Since the compressed size is compatible with the UDP tunnel MTU [11], fragmentation is not needed.

```
schc schc_send udp:51.91.121.182:23628 udp:54.37.158.10:8888
schc parser {('IPV6.VER', 1): [6, 4],
             ('IPV6.TC', 1): [0, 8],
             ('IPV6.FL', 1): [212084, 20],
             ('IPV6.LEN', 1): [9, 16, 'fixed'],
             ('IPV6.NXT', 1): [58, 8, 'fixed'],
             ('IPV6.HOP_LMT', 1): [52, 8, 'fixed'],
             ('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],
             ('IPV6.DEV_PREFIX', 1): [b' \x01\x04p\x1f!\x01\xd2', 64],
             ('IPV6.DEV_IID', 1): [b'\x00\x00\x00\x00\x00\x00\x00\x02', 64],
             ('ICMPV6.TYPE', 1): [128, 8],
             ('ICMPV6.CODE', 1): [0, 8],
             ('ICMPV6.CKSUM', 1): [40528, 16],
             ('ICMPV6.IDENT', 1): [382, 16],
             ('ICMPV6.SEQNO', 1): [1, 16]} b'\x00' None
schc compression rule {'RuleID': 6, 'RuleIDLength': 3, 'Compression': [
    {'FID': 'IPV6.VER', 'FL': 4, 'FP': 1, 'DI': 'BI', 'TV': 6, 'MO': 'EQUAL', 'CDA': 'NOT-SENT'},
    {'FID': 'IPV6.TC', 'FL': 8, 'FP': 1, 'DI': 'BI', 'TV': 0, 'MO': 'EQUAL', 'CDA': 'NOT-SENT'},
    {'FID': 'IPV6.FL', 'FL': 20, 'FP': 1, 'DI': 'BI', 'TV': 0, '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': 58, 'MO': 'EQUAL', 'CDA': 'NOT-SENT'},
    {'FID': 'IPV6.HOP_LMT', 'FL': 8, 'FP': 1, 'DI': 'BI', 'TV': 255, 'MO': 'IGNORE', 'CDA': 'NOT-SENT'},
    {'FID': 'IPV6.DEV_PREFIX', 'FL': 64, 'FP': 1, 'DI': 'BI', 'TV': b' \x01\x04p\x1f!\x01\xd2', 'MO': 'EQUAL', 'CDA': 'NOT-SENT'},
    {'FID': 'IPV6.DEV_IID', 'FL': 64, 'FP': 1, 'DI': 'BI', 'TV': b'\x00\x00\x00\x00\x00\x00\x00\x02', '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': 'DW', 'TV': 128, 'MO': 'EQUAL', 'CDA': 'NOT-SENT'},
    {'FID': 'ICMPV6.TYPE', 'FL': 8, 'FP': 1, 'DI': 'UP', 'TV': 129, 'MO': 'EQUAL', 'CDA': 'NOT-SENT'},
    {'FID': 'ICMPV6.CODE', 'FL': 8, 'FP': 1, 'DI': 'BI', 'TV': 0, 'MO': 'EQUAL', 'CDA': 'NOT-SENT'},
    {'FID': 'ICMPV6.CKSUM', 'FL': 16, 'FP': 1, 'DI': 'BI', 'TV': 0, 'MO': 'IGNORE', 'CDA': 'COMPUTE-CHECKSUM'},
    {'FID': 'ICMPV6.IDENT', 'FL': 16, 'FP': 1, 'DI': 'BI', 'TV': 0, 'MO': 'IGNORE', 'CDA': 'VALUE-SENT'},
    {'FID': 'ICMPV6.SEQNO', 'FL': 16, 'FP': 1, 'DI': 'BI', 'TV': 0, 'MO': 'IGNORE', 'CDA': 'VALUE-SENT'}],
    'META': {'UP_RULES': 15, 'DW_RULES': 15}}
schc compression result b'\xc4\x00\x28\x3a\x00\x60\x44\x40\x00\x00\x00\x00\x00\x00\x00\x02\x76\x60\x2f\xc0\x00\x20\x00'/171
protocol.py, schc_send, core_id:  udp:51.91.121.182:23628 device_id:  udp:54.37.158.10:8888 sender_delay 0 destination udp:54.37.158.10:8
schc fragmentation not needed size=171
protocol.py udp:54.37.158.10:8888
protocol.py (bytearray(b'\xc4\x00(:\x00‘D@\x00\x00\x00\x00\x00\x00\x02v‘/\xc0\x00 \x00'), 'udp:54.37.158.10:8888')
scapy_conection.py: send_pkt, dest  udp:54.37.158.10:8888 packet bytearray(b'\xc4\x00(:\x00‘D@\x00\x00\x00\x00\x00\x00\x02v‘/\xc0\x00 \x0
Queue running event -> 0, callback -> send_packet
```

11. 200 bits

Then the core gets the response from the device. Find the rule ID, apply decompression and returns a scapy structure that is displayed and sent.

```
tunneled SCHC msg
other end = udp:54.37.158.10:8888
uncomp_pkt ('udp:54.37.158.10:8888', <IPv6  version=6 tc=0 fl=0 nh=ICMPv6 hlim=255 src=2001:470:1f21:1d2::2 dst=2001:41d0:302:2200::13b3
id=0x17e seq=0x1 data=bytearray(b'\x00') |>>) type <class 'tuple'>
###[ IPv6 ]###
  version   = 6
  tc        = 0
  fl        = 0
  plen      = None
  nh        = ICMPv6
  hlim      = 255
  src       = 2001:470:1f21:1d2::2
  dst       = 2001:41d0:302:2200::13b3
###[ ICMPv6 Echo Reply ]###
     type      = Echo Reply
     code      = 0
     cksum     = None
     id        = 0x17e
     seq       = 0x1
     data      = bytearray(b'\x00')

WARNING: Incompatible L3 types detected using <class 'scapy.layers.inet6.IPv6'> instead of <class 'scapy.layers.inet.IP'> !
.
Sent 1 packets.
schc schc_send udp:51.91.121.182:23628 udp:54.37.158.10:8888
schc parser {('IPV6.VER', 1): [6, 4],
             ('IPV6.TC', 1): [0, 8],
             ('IPV6.FL', 1): [0, 20],
             ('IPV6.LEN', 1): [9, 16, 'fixed'],
             ('IPV6.NXT', 1): [58, 8, 'fixed'],
             ('IPV6.HOP_LMT', 1): [255, 8, 'fixed'],
             ('IPV6.APP_PREFIX', 1): [b' \x01\x04p\x1f!\x01\xd2', 64],
             ('IPV6.APP_IID', 1): [b'\x00\x00\x00\x00\x00\x00\x00\x02', 64],
             ('IPV6.DEV_PREFIX', 1): [b' \x01A\xd0\x03\x02"\x00', 64],
             ('IPV6.DEV_IID', 1): [b'\x00\x00\x00\x00\x00\x00\x13\xb3', 64],
             ('ICMPV6.TYPE', 1): [129, 8],
             ('ICMPV6.CODE', 1): [0, 8],
             ('ICMPV6.CKSUM', 1): [40272, 16],
             ('ICMPV6.IDENT', 1): [382, 16],
             ('ICMPV6.SEQNO', 1): [1, 16]} b'\x00' None
```

IPv6 packets that arrives to the SCHC core and are not matching any compression rule are rejected.

```
schc compression rule None
schc no-compression rule None
schc rule for compression/no-compression not found
protocol.py, schc_send, core_id:  udp:51.91.121.182:23628 device_id:  None sender_delay 0 destination udp:54.37.158.10:8888 position core
```

3.7.3 On the device

The program is launched, the same rules are loaded, the device ID is determined and when a rule 6/3 is received, this SCHC message is echoed.

```
\$ sudo python3 ping_device2.py
****************************************
Device: udp:54.37.158.10:8888
/-------------------------\
|Rule 6/3            110  |
|---------------+---+--+--+------------------------------+-------------+----------------\
|IPV6.VER       |  4| 1|BI|                             6|EQUAL        |NOT-SENT        |
|IPV6.TC        |  8| 1|BI|                             0|EQUAL        |NOT-SENT        |
|IPV6.FL        | 20| 1|BI|                             0|IGNORE       |NOT-SENT        |
|IPV6.LEN       | 16| 1|BI|------------------------------|IGNORE       |COMPUTE-LENGTH  |
|IPV6.NXT       |  8| 1|BI|                            58|EQUAL        |NOT-SENT        |
|IPV6.HOP_LMT   |  8| 1|BI|                           255|IGNORE       |NOT-SENT        |
|IPV6.DEV_PREFIX| 64| 1|BI|              200104701f2101d2|EQUAL        |NOT-SENT        |
|IPV6.DEV_IID   | 64| 1|BI|              0000000000000002|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|                           128|EQUAL        |NOT-SENT        |
|ICMPV6.TYPE    |  8| 1|UP|                           129|EQUAL        |NOT-SENT        |
|ICMPV6.CODE    |  8| 1|BI|                             0|EQUAL        |NOT-SENT        |
|ICMPV6.CKSUM   | 16| 1|BI|                             0|IGNORE       |COMPUTE-CHECKSUM|
|ICMPV6.IDENT   | 16| 1|BI|                             0|IGNORE       |VALUE-SENT      |
|ICMPV6.SEQNO   | 16| 1|BI|                             0|IGNORE       |VALUE-SENT      |
\---------------+---+--+--+------------------------------+-------------+----------------/
/-------------------------\
|Rule 12/11     00001100  |
```

```
!=======================+=============================================================\
!v Fragmentation mode : NoAck    header dtag 2 Window  0 FCN  3                    DW v!
!v No Tile size specified                                                          v!
!v RCS Algorithm: RCS_RFC8724                                                      v!
\=============================================================================================/
/-------------------------\
|Rule 13/11     00001101  |
!=======================+=============================================================\
!^ Fragmentation mode : NoAck    header dtag 2 Window  0 FCN  3                    UP ^!
!^ No Tile size specified                                                          ^!
!^ RCS Algorithm: RCS_RFC8724                                                      ^!
\=============================================================================================/
device_id is udp:54.37.158.10:8888
tunneled SCHC msg
ping
```

4. Fragmenting Ping6 with NoAck

In this chapter, in addition to compressing a ping6 request we will fragment it. For that, we will create a ping6 request that exceeds the maximum L2 MTU allowed for the device :

```
ping6 -c 1 -s 200 2001:470:1f21:1d2::2
```

```
##[ IPv6 ]##
    version   = 6
    tc        = 0
    fl        = 63154
    plen      = 58
    nh        = ICMPv6
    hlim      = 52
    src       = 2001:41d0:302:2200::13b3
    dst       = 2001:470:1f21:1d2::2
##[ ICMPv6 Echo Request ]##
       type      = Echo Request
       code      = 0
       cksum     = 0xbc64
       id        = 0x260
       seq       = 0x1
       data      = '\\xa5\\xf3\x1cb\x00\x00\x00\x00\x9d\r\x00\x00\x00\x00\x00\x10\x1 1\x12\x13\x14\x15\x16\x17\x18
                   \x19\x1a\x1b\x1c\x1d\x1e\x1f!"#$%&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`
                   abcdefghijklmnopqrstuvwxyz{|}~\x7f\\x80\\x81\\x82\\x83\\x84\\x85\\x86\\x87\\x88\\x89\\x8a\\x8b
                   \\x8c\\x8d\\x8e\\x8f\\x90\\x91\\x92\\x93\\x94\\x95\\x96\\x97\\x98\\x99\\x9a\\x9b\\x9c\\x9d\\x9e
                   \\x9f\\xa0\\xa1\\xa2\\xa3\\xa4\\xa5\\xa6\\xa7\\xa8\\xa9\\xaa\\xab\\xac\\xad\\xae\\xaf\\xb0\\xb1
                   \\xb2\\xb3\\xb4\\xb5\\xb6\\xb7\\xb8\\xb9\\xba\\xbb\\xbc\\xbd\\xbe\\xbf\\xc0\\xc1\\xc2\\xc3\\xc4
                   \\xc5\\xc6\\xc7''

    0000  FA 16 3E E9 DB 5D A2 C8 13 C9 D8 BC 08 00 45 00  ..>..]........E.
    0010  01 0C 79 DB 40 00 F0 29 33 33 D8 42 57 66 33 5B  ..y.@..)33.BWf3[
    0020  79 B6 60 00 F6 B2 00 D0 3A 34 20 01 41 D0 03 02  y.`.....:4 .A...
    0030  22 00 00 00 00 00 00 00 13 B3 20 01 04 70 1F 21  "......... ..p.!
    0040  01 D2 00 00 00 00 00 00 00 02 80 00 88 08 02 6B  ...............k
    0050  00 01 41 00 1D 62 00 00 00 00 14 3B 07 00 00 00  ..A..b.....;....
    0060  00 00 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D  ................
    0070  1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D  .. !"#$%&'()*+,-
    0080  2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D  ./0123456789:;<=
    0090  3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D  >?@ABCDEFGHIJKLM
    00a0  4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D  NOPQRSTUVWXYZ[\]
    00b0  5E 5F 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D  ^_`abcdefghijklm
    00c0  6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D  nopqrstuvwxyz{|}
    00d0  7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D  ~...............
    00e0  8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D  ................
    00f0  9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD  ................
    0100  AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD  ................
    0110  BE BF C0 C1 C2 C3 C4 C5 C6 C7                    ..........
```

As one may notice, in this case the size of the `data` field of the ICMPv6 Echo Request is

expanded in order to make the total package size larger. In this way, when the query arrives at the core it has to be first compressed and then fragmented so that it conforms to the MTU accepted by the device.

The following fragmentation rules are then added to the SoR and can be applied to that traffic :

Listing 4.1 – Fragmentation Rules in rule icmp2.json

```
        },{
                "RuleID" : 12,
                "RuleIDLength" : 11,
                "Fragmentation" : {
                        "FRMode": "NoAck",
                        "FRDirection": "DW"
        }
        },{
                "RuleID" : 13,
                "RuleIDLength" : 11,
                "Fragmentation" : {
                "FRMode": "NoAck",
                "FRDirection": "UP"
                }
    }
]
}
```

4.1 Understanding fragmentation rules

In addition to the rule ID and the rule ID length a fragmentation rule contains two parameters : the mode and the direction.

In order to support reliability, variable L2 MTUs and unidirectional links, the RFC 8724 defines three different fragmentation modes : (i) No-Ack, designed for limited and variable MTU sizes under the assumption that there is no out-of-sequence delivery, (ii) Ack-on-Error for variable MTU and out-of-order delivery using sporadic ACK messages and (iii) Ack-allways for invariable MTUs and no out-of-sequence delivery. In the following we will go deeper into details of these three modes.

As for the direction, it is necessary to stay the behaviour of the SCHC action based on where the traffic is originated. In our example, we will use the rule `12/11` for fragmentation on Downlink and rule `13/11` for fragmentation on Uplink. Therefore, if the traffic goes from the core to the device (Downlink), we use rule `12/11` for fragmenting the traffic at the core side and reassembling at the device side. On the contrary, rule `13/11` is used for fragmenting the traffic going from the device to the co and for the reassembly processes at the core side.

As stated in RFC 8724, in OpenSCHC, SCHC Fragmentation is always done after compression [1]. Then, as shown in Figure 4.1 the whole process goes as following :

1. It shall rather be noted that, the no-compression rule is also permitted. Therefore, if one is willing to use only SCHC Fragmentation, two rules should be defined : (i) no-compression rule and (ii) the desired fragmentation rule.

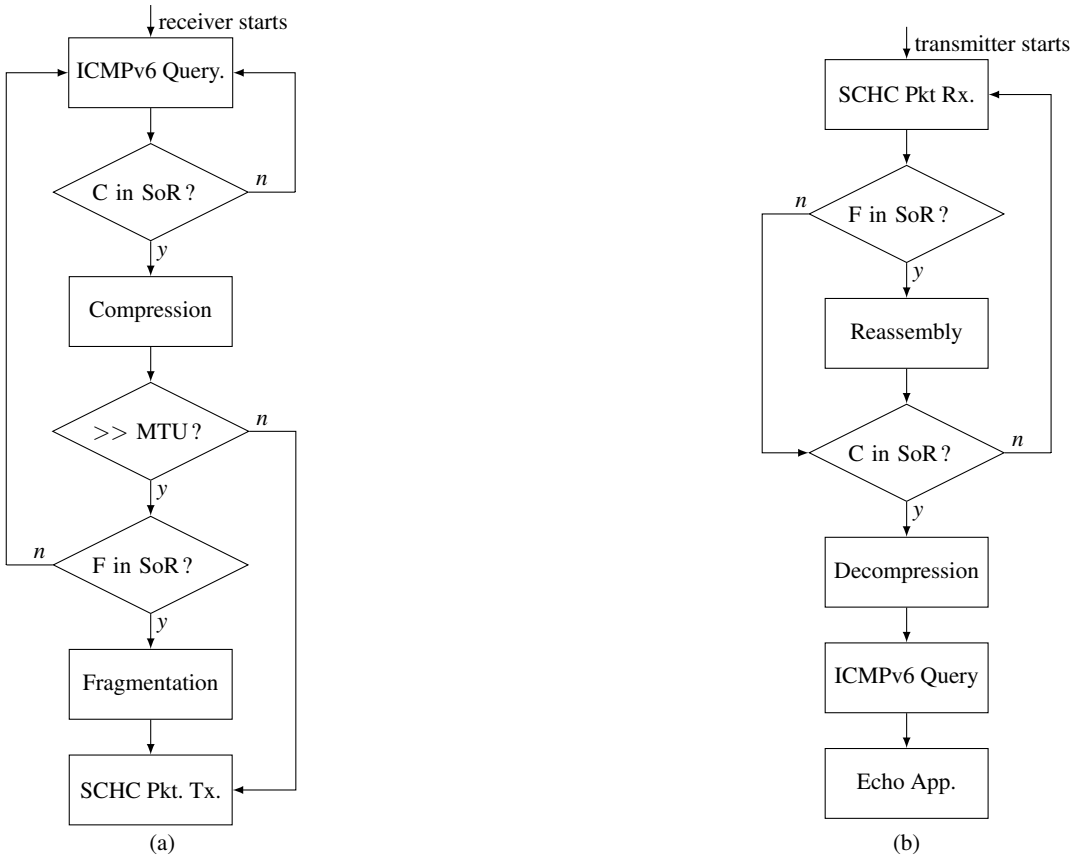

FIGURE 4.1 – ICMPv6 Query Reception when MTU exceeds L2 MTU : (a) Receiver behaviour, (b) Transmitter behaviour.

— ICMPv6 Echo Request Query Reception at core (Figure.4.1(a))
1. An user send a ICMPv6 Echo request exceeding the maximum MTU allowed.
2. The request arrives at the core side. It verifies if there is a compression rule on its SoR applicable to this specific traffic.
3. If it exists, the core applies the compression rule. Then, it verifies if the resulting packet surpasses the L2 MTU size.
4. If yes, the core verifies if there is a fragmentation rule on dowlink.
5. If there is one, the core applies the fragmentation using the mode defined in the rule.
6. The core starts to send fragments to the device.

— ICMPv6 Echo Request Query Reception at device (Figure.**??**(b)) :
1. The device receives SCHC packets containing the fragments.
2. The device verifies if there is a fragmentation rule and start the reassembly process.
3. Once the reassembly process is finished, the device search for a compression ruls applicable to this specific traffic.
4. The device applies the compression rule to decompress the packet.
5. The device retrieves the ICMPv6 Echo Reply Query.
6. At the application level, the device creates a ICMPv6 Echo Reply.

— ICMPv6 Echo Reply Query Transmission at device (Figure.**??**(a)) :
1. The device verifies if there is a Compression rule for the ICMPv6 Echo Reply.
2. If yes, the device compress the packet.
3. The device verifies if the resulting packet surpasses the L2 MTU size.
4. If yes, it looks for a Fragmentation rule in its SoR applicable to this traffic.
5. If there is one, the device applies the fragmentation using the mode as defined in the rule.
6. The device start to send fragments to the device.

— ICMPv6 Echo Reply Query Reception at core (Figure.**??**(b)) :
1. The core receives SCHC packets containing the fragments.
2. The core verifies if there is fragmentation rule in its SoR, if yes, it starts the reassembly process.
3. Once finished, the core search for compression rules applicable to this specific traffic
4. If there is a compression rue, it decompress the packet.
5. The core forwards the ICMPv6 Echo Reply to the user.

4.2 Fragmentation in No-ACK mode

This fragmentation mode is designed for no out-of sequence delivery and admits variable L2 MTU. In No-ACK mode, there is no communication from the fragment receiver to the fragment sender. The sender transmits all the SCHC Fragments without expecting any acknowledgement. Therefore, there is no need for bidirectional links.

4.2.1 SCHC Fragments Format

In No-ACK, there are two kinds of SCHC Fragments : (i) `All-0` fragments presented in Figure.4.2, and the last fragment called `All-1` depicted in Figure.4.3 An all-0 fragment, is composed of the following fields [2] :

2. The RFC 8724 also defines the `DTag` (Datagram Tag) and the `W` (Window) fields. The former is used for differentiating SCHC F/R messages belonging to different SCHC Packets, for our example this field is not present, and the latter, representing the Window size used in Ack-on-Error and Ack-Allways modes

— `RuleID`
— `FCN` : It is used to differentiate All-0 and All-1 fragments.
— `Fragment Payload` : Corresponds to the payload. Its size is aligned to the remaining space from to fit the L2 MTU.

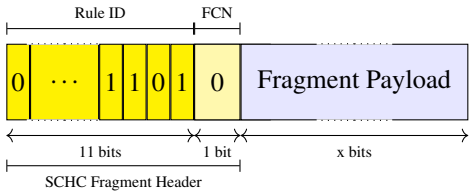

FIGURE 4.2 – All-0 SCHC Fragment in No-Ack mode

The second type of fragment is the `All-1`, it corresponds to the last fragment. As shown in Figure **??**, contrary to the `All-0`, it also contains the RCS field, and it can also contains padding as needed in order to fit the L2 word size.

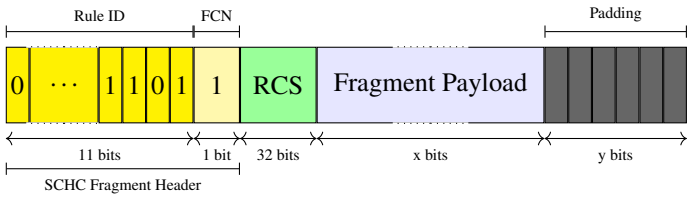

FIGURE 4.3 – All-1 SCHC Fragment in No-Ack mode

In OpenSCHC the Reassembly Check Sequence (RCS) field corresponds to the result of using the CRC32 algorithm, and as recommended by the RFC 8724 it is computed on the full SCHC packet (after reassembly) concatenated with the padding bits.

4.2.2 Fragmentation/Reassembly Process

In this mode, since there are no fragment acknowledgments, the sender creates as many fragments as needed based on the size of the compressed SCHC packet and the L2 MTU. Figure 4.4 presents an example where n fragments are needed. In this case, the transmitter creates $n-1$ `All-0` fragments and one `All-1` with the corresponding RCS field and padding if needed.

4.2.3 The code

In this section, we will present the code necessary to process an ICMPv6 echo request that exceeds the L2 MTU size. Two blocks of code are required : core and device programs. For the downlink, they are in charge of :

— core program : receiving the IPv6 packet, compressing it, fragmenting it and sending the SCHC fragments.
— device program : reassembling and decompressing SCHC packets and, creating the echo reply packet.

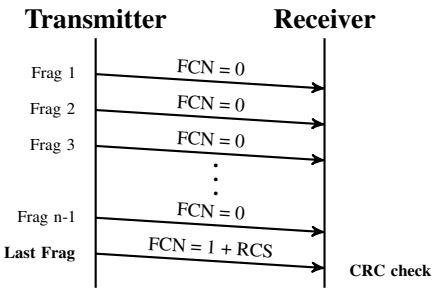

FIGURE 4.4 – All-1 SCHC Fragment in No-Ack mode

For the uplink, they are in charge of :

— device program : compressing and fragmenting the IPv6 Echo Reply packet, and sending the SCHC fragments to the core.

— core program : reassembling and decompressing SCHC packets and, forwarding the Echo reply IPv6 Packet to the user.

Core program

For this example we will extend the code used in Section 3.4.1. As in the previous case, the Compression and Fragmentation processes start with the creation of a Rule Manager «`rm`» used to add the rules from the file `icmp2.json` (cf. 4.1 on page 32), which includes the Rule 3 for compressing and Rules 12 and 13 for fragmenting.

As presented in Listing **??** on page ??, the process follows the same logic as in the compression example. The `processPkt` function filters SCHC packets coming from devices and IPv6 tunneled packets from Hurricane Electric.

As in the C/D only example, the `processPkt` calls the SCHC Machine and then looks at the received packets. There can be two kinds off packets : (i) IPv6 tunneled packets filtered by looking at the IP proto 41 and passed to the SCHC Machine removing the first 34 bytes (Ethernet and IPv4 headers), and (ii) SCHC Packets filtered by looking at the UDP socket port `0x5C4C`.

Listing 4.2 – ping_core2.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 scapy_connection import *
from gen_utils import dprint, sanitize_value
from scapy.layers.inet import IP
import pprint
import binascii
import socket
import ipaddress

```

```
# Create a Rule Manager and upload the rules.
rm = RM.RuleManager()
rm.Add(file="icmp2.json")
rm.Print()

def processPkt(pkt):
    """ called when scapy receives a packet, since this function takes only one argument,
    schc_machine and scheduler must be specified as a global variable.
    """
    scheduler.run(session=schc_machine)
    # look for a tunneled SCHC pkt
    if pkt.getlayer(Ether) != None: #HE tunnel do not have Ethernet
        e_type = pkt.getlayer(Ether).type
        if e_type == 0x0800:
            ip_proto = pkt.getlayer(IP).proto
            if ip_proto == 17:
                udp_dport = pkt.getlayer(UDP).dport
                if udp_dport == socket_port: # tunnel SCHC msg to be decompressed
                    print ("tunneled SCHC msg")
                    schc_pkt, addr = tunnel.recvfrom(2000)
                    other_end = "udp:"+addr[0]+":"+str(addr[1])
                    print("other end =", other_end)
                    uncomp_pkt = schc_machine.schc_recv(device_id=other_end,
                                    schc_packet=schc_pkt)
                    if uncomp_pkt != None:
                        uncomp_pkt[1].show()
                        send(uncomp_pkt[1], iface="he-ipv6")
            elif ip_proto==41:
                schc_machine.schc_send(bytes(pkt)[34:])

# Start SCHC Machine
POSITION = T_POSITION_CORE

socket_port = 0x5C4C
tunnel = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
tunnel.bind(("0.0.0.0", socket_port))

lower_layer = ScapyLowerLayer(position=POSITION, socket=tunnel, other_end=None)
system = ScapySystem()
scheduler = system.get_scheduler()
schc_machine = SCHCProtocol(
    system=system,           # define the scheduler
    layer2=lower_layer,      # how to send messages
    role=POSITION,           # DEVICE or CORE
    verbose = True)
schc_machine.set_rulemanager(rm)

sniff(prn=processPkt, iface="ens3") # scappy cannot read multiple interfaces
```

4.2.4 The execution

Start the device and core programs, is sudo mode :

```
$ sudo python3.9 ping_core2.py
```

```
$ sudo python3.9 ping_device2.py
```

The program will display the rules and cursor spins, indicating the SCHC machine is running. As we can see, three rules are defined : Rule 6/3 for compression, Rule 12/11 for fragmentation in Uplink using No-Ack mode and Rule 13/11 using No-Ack mode.

```
**************************************
Device: udp:54.37.158.10:8888
/-------------------------\
|Rule 6/3             110  |
|---------------+---+--+--+------------------------------+-------------+----------------\
|IPV6.VER       |  4| 1|BI|                             6|EQUAL        |NOT-SENT
|
|IPV6.TC        |  8| 1|BI|                             0|EQUAL        |NOT-SENT
|
|IPV6.FL        | 20| 1|BI|                             0|IGNORE       |NOT-SENT
|
|IPV6.LEN       | 16| 1|BI|------------------------------|IGNORE       |COMPUTE-LENGTH
|
|IPV6.NXT       |  8| 1|BI|                            58|EQUAL        |NOT-SENT
|
|IPV6.HOP_LMT   |  8| 1|BI|                           255|IGNORE       |NOT-SENT
|
|IPV6.DEV_PREFIX| 64| 1|BI|              200104701f2101d2|EQUAL        |NOT-SENT
|
|IPV6.DEV_IID   | 64| 1|BI|              0000000000000002|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|                           128|EQUAL        |NOT-SENT
|
|ICMPV6.TYPE    |  8| 1|UP|                           129|EQUAL        |NOT-SENT
|
|ICMPV6.CODE    |  8| 1|BI|                             0|EQUAL        |NOT-SENT
|
|ICMPV6.CKSUM   | 16| 1|BI|                             0|IGNORE       |COMPUTE-CHECKSUM|
|ICMPV6.IDENT   | 16| 1|BI|                             0|IGNORE       |VALUE-SENT
|
|ICMPV6.SEQNO   | 16| 1|BI|                             0|IGNORE       |VALUE-SENT
|
\---------------+---+--+--+------------------------------+-------------+----------------/
/-------------------------\
|Rule 12/11     00001100  |
!=========================+=========================================================================\
!v Fragmentation mode : NoAck    header dtag 2 Window  0 FCN  3                      DW v!
!v No Tile size specified
v!
!v RCS Algorithm: crc32
v!
\=========================================================================================/
/-------------------------\
|Rule 13/11     00001101  |
!=========================+=========================================================================\
!^ Fragmentation mode : NoAck    header dtag 2 Window  0 FCN  3                      UP ^!
!^ No Tile size specified
^!
!^ RCS Algorithm: crc32
^!
\=========================================================================================/
/
```

Then, we start pinging the device defined in the SoR, the `-c 1` limits the number of ping messages, and `-s 50` increases the size of the ICMPv6 packet.

```
$ ping6 -c 1 -s 50 dev2.openschc.net
```

And we get as result :

```
58 bytes from 2001:470:1f21:1d2::2 (2001:470:1f21:1d2::2): icmp_seq=1 ttl=239 time=136 ms

--- dev2.openschc.net ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 135.661/135.661/135.661/0.000 ms
```

The core instance displays :

```
schc recv-from-l3 None None
schc parser {('IPV6.VER', 1): [6, 4],  ('IPV6.TC', 1): [0, 8],  ('IPV6.FL', 1): [749668, 20],
            ('IPV6.LEN', 1): [58, 16, 'fixed'], ('IPV6.NXT', 1): [58, 8, 'fixed'],
            ('IPV6.HOP_LMT', 1): [46, 8, 'fixed'],
            ('IPV6.APP_PREFIX', 1): [b'*\x01\x0e\n\x01\xa6\x1c\xe0', 64],
            ('IPV6.APP_IID', 1): [b'\xd6\x0fN\x05\xd1z\x14\xcd', 64],
            ('IPV6.DEV_PREFIX', 1): [b' \x01\x04p\x1f!\x01\xd2', 64],
            ('IPV6.DEV_IID', 1): [b'\x00\x00\x00\x00\x00\x00\x00\x02', 64],
            ('ICMPV6.TYPE', 1): [128, 8], ('ICMPV6.CODE', 1): [0, 8], ('ICMPV6.CKSUM', 1): [26192, 16],
            ('ICMPV6.IDENT', 1): [2, 16], ('ICMPV6.SEQNO', 1): [1, 16]}

schc compression rule {'RuleID': 6, 'RuleIDLength': 3,
                       'Compression': [{'FID': 'IPV6.VER', 'FL': 4, 'FP': 1, 'DI': 'BI', 'TV': 6, 'MO': 'EQUAL',
                       'CDA': 'NOT-SENT'}, ... }

schc compression result b\xc5\x40\x21\xc1\x40\x34\xc3\x9c\x1a\xc1\xe9\xc0\xba\x2f\x42\x99\xa0\x00\x40\x00\x31\x38
                         \x8a\xcc\x40\x00\x00\x00\x0d\x51\x20\xc0\x00\x00\x00\x00\x02\x02\x22\x42\x62\x82\xa2\xc2
                         \xe3\x03\x23\x43\x63\x83\xa3\xc3\xe4\x04\x24\x44\x64\x84\xa4\xc4\xe5\x05\x25\x45\x65\x85
                         \xa5\xc5\xe6\x06\x20/563

schc fragmentation rule {'RuleID': 12, 'RuleIDLength': 11, 'Fragmentation': {'FRDirection': 'DW', 'FRMode': 'NoAck', ...}

MTU =  56

r:12/11 (noA) DTAG=0 W=- FCN=All-0
|---- 25------------>
frag_send.py, args: (bytearray(b'\x01\x80\xc5@!\xc1@4\xc3\x9c\x1a\xc1\xe9\xc0\xba/B\x99\xa0\x00@\x0018\x8a'),
                    'udp:54.37.158.10:8888', None)

Queue running event -> 1, callback -> send_packet
MTU =  56
r:12/11 (noA) DTAG=0 W=- FCN=All-0
|---- 25------------>
frag_send.py, args: (bytearray(b'\x01\x80\xcc@\x00\x00\x00\rQ \xc0\x00\x00\x00\x00\x02\x02"Bb\x82\xa2\xc2\xe3\x03'),
                    'udp:54.37.158.10:8888', None)
Queue running event -> 2, callback -> send_frag
Queue running event -> 3, callback -> event_sent_frag
Queue running event -> 4, callback -> send_packet
MTU =  56
r:12/11 (noA) DTAG=0 W=- FCN=All-0
|---- 25------------>
frag_send.py, args: (bytearray(b'\x01\x80#Cc\x83\xa3\xc3\xe4\x04$Dd\x84\xa4\xc4\xe5\x05%Ee\x85\xa5\xc5\xe6'),
                    'udp:54.37.158.10:8888', None)
frag_send.py, _session_id:  udp:54.37.158.10:8888
Queue running event -> 5, callback -> send_frag
Queue running event -> 6, callback -> event_sent_frag
Queue running event -> 7, callback -> send_packet
MTU =  56
SessionManager: deleted ('udp:54.37.158.10:8888', 12, 11, 0)
MIC Size =  32
r:12/11 (noA) DTAG=0 W=- FCN=All-1
|----  8------------>
frag_send.py, args:  (bytearray(b'\x01\x87x\x10\xdc_\x06 '), 'udp:54.37.158.10:8888', None)
frag_send.py, _session_id:  udp:54.37.158.10:8888
Queue running event -> 8, callback -> send_frag
Queue running event -> 9, callback -> send_packet
tunneled SCHC msg
other end = udp:54.37.158.10:8888
                         ----------- 25--------->|
New reassembly session created ReassemblerNoAck
frag data
                         r:13/11 (noA) DTAG=0 W=- FCN=All-0
CANCEL Inactivity Timer None
[b'\xc5\x40\x21\xc1\x40\x34\xc3\x9c\x1a\xc1\xe9\xc0\xba\x2f\x42\x99\xa0\x00\x40\x00\x31\x38\x8a'/184]
tunneled SCHC msg
other end = udp:54.37.158.10:8888
                         ----------- 25--------->|
Reassembly session found ReassemblerNoAck
frag data
                         r:13/11 (noA) DTAG=0 W=- FCN=All-0
CANCEL Inactivity Timer 10
[b'\xc5\x40\x21\xc1\x40\x34\xc3\x9c\x1a\xc1\xe9\xc0\xba\x2f\x42\x99\xa0\x00\x40\x00\x31\x38\x8a'/184,
b'\xcc\x40\x00\x00\x00\x0d\x51\x20\xc0\x00\x00\x00\x00\x02\x02\x22\x42\x62\x82\xa2\xc2\xe3\x03'/184]
```

```
tunneled SCHC msg
other end = udp:54.37.158.10:8888
                        ----------- 25--------->|
Reassembly session found ReassemblerNoAck
frag data
                        r:13/11 (noA) DTAG=0 W=- FCN=All-0
CANCEL Inactivity Timer 11
[b'\xc5\x40\x21\xc1\x40\x34\xc3\x9c\x1a\xc1\xe9\xc0\xba\x2f\x42\x99\xa0\x00\x40\x00\x31\x38\x8a'/184,
 b'\xcc\x40\x00\x00\x00\x0d\x51\x20\xc0\x00\x00\x00\x00\x02\x02\x22\x42\x62\x82\xa2\xc2\xe3\x03'/184,
 b'\x23\x43\x63\x83\xa3\xc3\xe4\x04\x24\x44\x64\x84\xa4\xc4\xe5\x05\x25\x45\x65\x85\xa5\xc5\xe6'/184]
tunneled SCHC msg
other end = udp:54.37.158.10:8888
                        -----------  8--------->|
Reassembly session found ReassemblerNoAck
frag data
                        r:13/11 (noA) DTAG=0 W=- FCN=All-1
CANCEL Inactivity Timer 12
[b'\xc5\x40\x21\xc1\x40\x34\xc3\x9c\x1a\xc1\xe9\xc0\xba\x2f\x42\x99\xa0\x00\x40\x00\x31\x38\x8a'/184,
 b'\xcc\x40\x00\x00\x00\x0d\x51\x20\xc0\x00\x00\x00\x00\x02\x02\x22\x42\x62\x82\xa2\xc2\xe3\x03'/184,
 b'\x23\x43\x63\x83\xa3\xc3\xe4\x04\x24\x44\x64\x84\xa4\xc4\xe5\x05\x25\x45\x65\x85\xa5\xc5\xe6'/184,
 b'\x06\x20'/16]
----------------------- Final Reassembly -----------------------
ALL1 received
b'c54021c14034c39c1ac1e9c0ba2f4299a000400031388acc400000000d5120c0000000000020222426282a2c2e30323436
  383a3c3e40424446484a4c4e50525456585a5c5e60620'

SUCCESS: MIC matched. packet bytearray(b'x\x10\xdc_') == result b'x\x10\xdc_'
----------------------- Decompression -----------------------
SessionManager: deleted (None, 13, 11, 0)
###[ IPv6 ]###
  version   = 6
  tc        = 0
  fl        = 0
  plen      = None
  nh        = ICMPv6
  hlim      = 255
  src       = 2001:470:1f21:1d2::2
  dst       = 2a01:e0a:1a6:1ce0:d60f:4e05:d17a:14cd
###[ ICMPv6 Echo Reply ]###
     type      = Echo Reply
     code      = 0
     cksum     = None
     id        = 0x2
     seq       = 0x1
     data      = bytearray(b'\x89\xc4Vb\x00\x00\x00\x00j\x89\x06\x00\x00\x00\x00\x00\x10\x11\x12\x13...')

.
Sent 1 packets.
schc recv-from-l3 None None
schc parser {('IPV6.VER', 1): [6, 4], ('IPV6.TC', 1): [0, 8],
             ('IPV6.FL', 1): [0, 20], ('IPV6.LEN', 1): [58, 16, 'fixed'],
             ('IPV6.NXT', 1): [58, 8, 'fixed'], ('IPV6.HOP_LMT', 1): [255, 8, 'fixed'],
             ('IPV6.APP_PREFIX', 1): [b' \x01\x04p\x1f!\x01\xd2', 64],
             ('IPV6.APP_IID', 1): [b'\x00\x00\x00\x00\x00\x00\x00\x02', 64],
             ('IPV6.DEV_PREFIX', 1): [b'*\x01\x0e\n\x01\xa6\x1c\xe0', 64],
             ('IPV6.DEV_IID', 1): [b'\xd6\x0fN\x05\xd1z\x14\xcd', 64],
             ('ICMPV6.TYPE', 1): [129, 8], ('ICMPV6.CODE', 1): [0, 8],
             ('ICMPV6.CKSUM', 1): [25936, 16], ('ICMPV6.IDENT', 1): [2, 16],
             ('ICMPV6.SEQNO', 1): [1, 16]}

schc compression rule None
schc no-compression rule None
schc rule for compression/no-compression not found
\
.
Sent 1 packets.
```

We can see several steps. The following are those corresponding to the C/F Process one the ICMPv6 message arrives to the core instance :

- Parse the packet : from a sequence of bytes received on the network, create a list of fields containing the field identification and their associated value.
- Find a valid compression rule : ask the rule manager to find a rule matching the parsed packet. The rule selection will also provide the device ID.
- The Rule Manager finds a Compression rule and the SCHC Machine applies the compression rule.
- The SCHC Machine verifies if the MTU is below the size of the compressed packet, and if not it looks for a fragmentation rule for this packet.

— The Rule Manager finds the Rule 12 that correspond to fragmentation in No-Ack mode
— The SCHC Machine creates a context used to track the fragmentation session. It corresponds to the Rule ID, the Rule ID length and the dtag. In our example dtag is set to zero.
— The SCHC Machine starts to create fragments and sent it into the Network. In this example, three All-0 fragments and 1 All-1 Fragment are created.
— Fragments are sent on the UDP tunnel using the corresponding device ID `udp:54.37.158.10:8888`

Then, once the device has performed the R/D process, created the the echo query response and C/F the prompt shows the following messages :

— Scappy sniffs an udp packet and detects that it comes from the device by looking at the IP and corresponding port : `other_end = udp:54.37.158.10:8888`.
— The SCHC Machine creates a reassembly session using the NoAck mode.
— The SCHC Machine stores all the fragments until the All-1 is received.
— The SCHC Machine reassembles the fragments and decompress it.
— Once decompressed, the received packet is printed using scapy ; we can see that it correspond to the ICMP Echo Query Response created by the device.
— Finally, the core sends the IPv6 packet to the user

At the device side we get the following messages :

```
tunneled SCHC msg
                        ----------- 25--------->|
New reassembly session created ReassemblerNoAck
frag data
                        r:12/11 (noA) DTAG=0 W=- FCN=All-0
CANCEL Inactivity Timer None
[b'\xc5\x40\x21\xc1\x40\x34\xc3\x9c\x1a\xc1\xe9\xc0\xba\x2f\x42\x99\xa0\x00\x40\x00\x31\x38\x8a'/184]
tunneled SCHC msg
                        ----------- 25--------->|
Reassembly session found ReassemblerNoAck
frag data
                        r:12/11 (noA) DTAG=0 W=- FCN=All-0
CANCEL Inactivity Timer 0
[b'\xc5\x40\x21\xc1\x40\x34\xc3\x9c\x1a\xc1\xe9\xc0\xba\x2f\x42\x99\xa0\x00\x40\x00\x31\x38\x8a'/184,
 b'\xcc\x40\x00\x00\x00\x0d\x51\x20\xc0\x00\x00\x00\x00\x02\x02\x22\x42\x62\x82\xa2\xc2\xe3\x03'/184]
tunneled SCHC msg
                        ----------- 25--------->|
Reassembly session found ReassemblerNoAck
frag data
                        r:12/11 (noA) DTAG=0 W=- FCN=All-0
CANCEL Inactivity Timer 1
[b'\xc5\x40\x21\xc1\x40\x34\xc3\x9c\x1a\xc1\xe9\xc0\xba\x2f\x42\x99\xa0\x00\x40\x00\x31\x38\x8a'/184,
 b'\xcc\x40\x00\x00\x00\x0d\x51\x20\xc0\x00\x00\x00\x00\x02\x02\x22\x42\x62\x82\xa2\xc2\xe3\x03'/184,
 b'\x23\x43\x63\x83\xa3\xc3\xe4\x04\x24\x44\x64\x84\xa4\xc4\xe5\x05\x25\x45\x65\x85\xa5\xc5\xe6'/184]
tunneled SCHC msg
                        -----------  8--------->|
Reassembly session found ReassemblerNoAck
frag data
                        r:12/11 (noA) DTAG=0 W=- FCN=All-1
CANCEL Inactivity Timer 2
[b'\xc5\x40\x21\xc1\x40\x34\xc3\x9c\x1a\xc1\xe9\xc0\xba\x2f\x42\x99\xa0\x00\x40\x00\x31\x38\x8a'/184,
 b'\xcc\x40\x00\x00\x00\x0d\x51\x20\xc0\x00\x00\x00\x00\x02\x02\x22\x42\x62\x82\xa2\xc2\xe3\x03'/184,
 b'\x23\x43\x63\x83\xa3\xc3\xe4\x04\x24\x44\x64\x84\xa4\xc4\xe5\x05\x25\x45\x65\x85\xa5\xc5\xe6'/184,
 b'\x06\x20'/16]
----------------------- Final Reassembly -----------------------
ALL1 received
decompressed pkt: b'c54021c14034c39c1ac1e9c0ba2f4299a000400031388acc400000000d5120c0000000000202224262
83a3c3e40424446484a4c4e50525456585a5c5e60620'
SUCCESS: MIC matched. packet bytearray(b'x\x10\xdc_') == result b'x\x10\xdc_'
----------------------- Decompression -----------------------
SessionManager: deleted (None, 12, 11, 0)
###[ IPv6 ]###
  version   = 6
  tc        = 0
  fl        = 0
```

```
  plen      = None
  nh        = ICMPv6
  hlim      = 255
  src       = 2001:470:1f21:1d2::2
  dst       = 2a01:e0a:1a6:1ce0:d60f:4e05:d17a:14cd
###[ ICMPv6 Echo Reply ]###
     type      = Echo Reply
     code      = 0
     cksum     = None
     id        = 0x2
     seq       = 0x1
     data      = '\\x89\\xc4Vb\x00\x00\x00\x00j\\x89\x06\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19
\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01'

schc recv-from-l3 udp:51.91.121.182:23628 None
schc parser {('IPV6.VER', 1): [6, 4],
             ('IPV6.TC', 1): [0, 8],
             ('IPV6.FL', 1): [0, 20],
             ('IPV6.LEN', 1): [58, 16, 'fixed'],
             ('IPV6.NXT', 1): [58, 8, 'fixed'],
             ('IPV6.HOP_LMT', 1): [255, 8, 'fixed'],
             ('IPV6.DEV_PREFIX', 1): [b' \x01\x04p\x1f!\x01\xd2', 64],
             ('IPV6.DEV_IID', 1): [b'\x00\x00\x00\x00\x00\x00\x00\x02', 64],
             ('IPV6.APP_PREFIX', 1): [b'*\x01\x0e\n\x01\xa6\x1c\xe0', 64],
             ('IPV6.APP_IID', 1): [b'\xd6\x0fN\x05\xd1z\x14\xcd', 64],
             ('ICMPV6.TYPE', 1): [129, 8], ('ICMPV6.CODE', 1): [0, 8],
             ('ICMPV6.CKSUM', 1): [25936, 16],
             ('ICMPV6.IDENT', 1): [2, 16],
             ('ICMPV6.SEQNO', 1): [1, 16]}

schc compression rule {'RuleID': 6, 'RuleIDLength': 3,
                       'Compression': [{'FID': 'IPV6.VER', 'FL': 4, 'FP': 1, 'DI': 'BI', 'TV': 6, 'MO': 'EQUAL',
                       'CDA': 'NOT-SENT'}, {'FID': 'IPV6.TC' ...]

schc compression result b\xc5\x40\x21\xc1\x40\x34\xc3\x9c\x1a\xc1\xe9\xc0\xba\x2f\x42\x99\xa0\x00\x40\x00\x31\x38
                          \x8a\xcc\x40\x00\x00\x00\x0d\x51\x20\xc0\x00\x00\x00\x00\x02\x02\x22\x42\x62\x82\xa2\xc2
                          \xe3\x03\x23\x43\x63\x83\xa3\xc3\xe4\x04\x24\x44\x64\x84\xa4\xc4\xe5\x05\x25\x45\x65\x85
                          \xa5\xc5\xe6\x06\x20/563

schc fragmentation rue {'RuleID': 13, 'RuleIDLength': 11, 'Fragmentation': {'FRDirection': 'UP', 'FRMode': 'NoAck'...}}
MTU =  56
r:13/11 (noA) DTAG=0 W=- FCN=All-0
|---- 25------------>
frag_send.py, args: (bytearray(b'\x01\xa0\xc5@!\xc1@4\xc3\x9c\x1a\xc1\xe9\xc0\xba/B\x99\xa0\x00@\x0018\x8a'),
                     'udp:51.91.121.182:23628', None)
frag_send.py, _session_id:  udp:51.91.121.182:23628
FCN size= 0
Queue running event -> 3, callback -> event_sent_frag
scapy_conection.py: send_pkt, dest  udp:51.91.121.182:23628 packet bytearray(b'\x01\...')
Queue running event -> 4, callback -> send_packet
MTU =  56
r:13/11 (noA) DTAG=0 W=- FCN=All-0
|---- 25------------>
frag_send.py, args:  (bytearray(b'\x01...'), 'udp:51.91.121.182:23628', None)
frag_send.py, _session_id:  udp:51.91.121.182:23628
Queue running event -> 5, callback -> send_frag
event_sent_frag
Queue running event -> 6, callback -> event_sent_frag
scapy_conection.py: send_pkt, dest  udp:51.91.121.182:23628 packet bytearray(b'\x01\...')
Queue running event -> 7, callback -> send_packet
MTU =  56
r:13/11 (noA) DTAG=0 W=- FCN=All-0
|---- 25------------>
frag_send.py, args:  (bytearray(b'\x01\...')
frag_send.py, _session_id:  udp:51.91.121.182:23628
Queue running event -> 8, callback -> send_frag
event_sent_frag
Queue running event -> 9, callback -> event_sent_frag
scapy_conection.py: send_pkt, dest  udp:51.91.121.182:23628 packet bytearray(b'\x01\x....')
Queue running event -> 10, callback -> send_packet
MTU =  56
SessionManager: deleted ('udp:51.91.121.182:23628', 13, 11, 0)
r:13/11 (noA) DTAG=0 W=- FCN=All-1
|----  8------------>
frag_send.py, args:  (bytearray(b'\x01\xa7x\x10\xdc_\x06 '), 'udp:51.91.121.182:23628', None)
frag_send.py, _session_id:  udp:51.91.121.182:23628
Queue running event -> 11, callback -> send_frag
scapy_conection.py: send_pkt, dest  udp:51.91.121.182:23628 packet bytearray(b'\x01\xa7x\x10\xdc_\x06')
Queue running event -> 12, callback -> send_packet
```

We can tell that, the process follows the following steps :

— Reception of a SCHC Packet

— The SCHC Machine detects that it corresponds to a SCHC Fragment and creates a reassembly session and stores the fragments as they arrive.
— Once the All-1 arrives, the SCHC Machine reassembles the packet.
— The SCHC Machine validates the CRC.
— The Echo Reply is created.
— As in the core side, the Rule Manager finds a compression rule, then, the packet is parsed, compressed and fragmented.
— Three All-0 and one All-1 SCHC fragments are created and sent back to the core.

Finally, sudo tcpdump -nXi ens3 udp port 0x5C4C at the core side. As we can see SCHC packets are first sent on the tcp tunnel and

5. Answers to the questions

Index

H

I

L

M

N

P

R

S

T

U